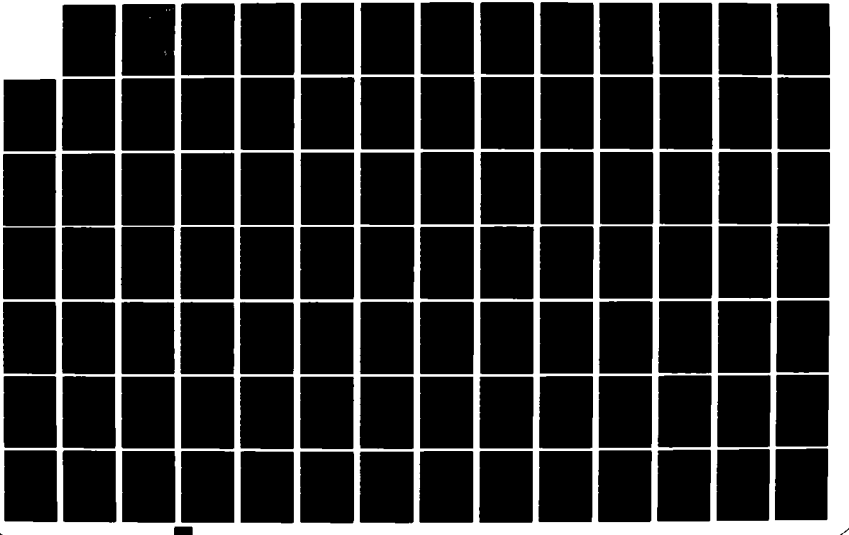
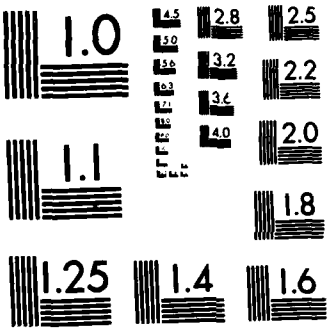
AD-A124 927 THE CONTINUED DESIGN AND IMPLEMENTATION OF A RELATIONAL 1/2
DATABASE SYSTEM(U) AIR FORCE INST OF TECH
WRIGHT-PATTERSON AFB OH SCHOOL OF ENGINEERING
UNCLASSIFIED L M RODGERS DEC 82 AFIT/GCS/EE/82D-29 F/G 9/2 NL





THE CONTINUED DESIGN AND IMPLEMENTATION OF A RELATIONAL DATABASE SYSTEM

THESIS

AFIT/GCS/EE/82 Linda M. Rodgers
2Lt USAF

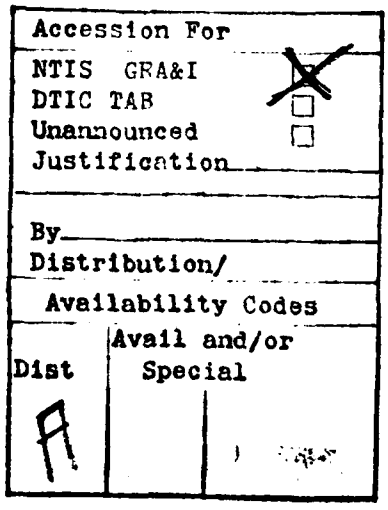
DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY (ATC)
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

83 02 024 035

AFIT/GCS/EE/82-29

Dr. Hartrum
Dr. Lamont
Maj. Varrieur



THE CONTINUED DESIGN AND IMPLEMENTATION

OF A

RELATIONAL DATABASE SYSTEM

THESIS

AFIT/GCS/EE/82 Linda M. Rodgers
2Lt USAF

Approved for public release; distribution unlimited....

AFIT/GCS/EE/82D-29

THE CONTINUED DESIGN AND IMPLEMENTATION OF A RELATIONAL DATABASE SYSTEM

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University (ATC)
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science

by

Linda M. Rodgers
2Lt USAF

Graduate Computer Systems

December 1982

Preface

The desire for a teaching aid and an in house research tool in the area of database systems has been expressed by several AFIT Faculty members. Specifically, in 1979, Dr. Thomas Hartrum, on the faculty of the AFIT/EN Electrical Engineering Department, proposed that such a system be developed as a master's thesis. The initial development of such a tool was undertaken by 2Lt. Mark A. Roth, and at the completion of his thesis effort, he left a design and partial implementation for the front end of such a system. In addition, he left some suggestions and recommendations for the continued development of this system. I undertook this project with the intention of completing the design and implementation of this system based on the recommendations suggested by Mark A. Roth. Specifically, this included completing the implementation of the edit modules, the run modules, and the low level access mechanism.

Although this was the initial goal, it was determined that expansive development may need to be considered on a different system which could provide more memory or under a different version of the Pascal Operating System which would allow for more segmentation. As a consequence, the approach decided upon during this thesis effort was to design and implement an algorithm for use with the low

level access structure and also to continue the implementation of the front end system. Hopefully, from the design and partial implementation of this access structure, the completion of this system can be realized.

I would like to give my thanks to Dr. Hartrum, Dr. Lamont and Maj. Varrieur for their comments and encouragement during the continued development of this project and in the development and writing of this document.

Contents

LIST OF FIGURES

Abstract

The Roth Relational Database System is a database system designed and implemented for teaching and research purposes at the Air Force Institute of Technology. The system was originally designed and partially implemented by 2Lt. Mark A. Roth in 1979 and continued design and implementation has been accomplished by James Mau. During this thesis effort, an investigation into the design and implementation of the Roth Relational Database System is made in order to continue the design and implementation of the AFIT Relational Database System. Additional research was done to assess the feasibilty of the suggestions made by Roth concerning the continued development of the system. Specifically, an investigation was made concerning Theo Haerder's access structure and an investigation was made concerning the interface between the editor and the low level access mechanism.

Continuing this project, a top down structured design was completed for the edit functions and the low level access structure. Once this was accomplished, an algorithm was developed, implemented and tested for the insertion and deletion of values into a B-tree.

THE CONTINUED DESIGN AND IMPLEMENTATION OF A RELATIONAL DATABASE SYSTEM

I. INTRODUCTION

BACKGROUND

In recent years, the use of database management systems for information processing has increased. Not only has automated information processing made it easier to access an increasing wealth of information, but it has made it less complicated to maintain this information from a user's point of view. In addition, the introduction of a database management system may provide centralized control for an organization. However, it is the case as with any new technology, that several improvements still need to be made. Since the installation of such systems has proved beneficial, there is an increasing demand for its use. However, the level of efficiency that can be obtained by such existing systems has been limited because of the increasing amounts of information to be stored and the need to improve the methods for maintaining and accessing this information. Hence, research continues so as to improve these systems. Such research is in the area of database software.

In the area of improving computer software, emphasis is on the improvement of algorithm design and the improvement of file organization. Such improvements are

needed in order to improve processing time and memory usage. As a consequence, the overall efficiency of a computer system can be greatly increased.

One way of achieving these improvements is found in a particular area of software research. This area is the optimization of information query processing. In particular, research in this area has focused on the relational model because of its characteristic features. Specifically, the relational model offers a much simpler view of the data to the user. Also, it enforces data independence and has the potential to limit data redundancy. In terms of design, the relational approach offers an additional advantage. The relational view tends to hide actual implementation concerns from the user. As a consequence, a more structured approach can be taken to the design of the database system. This is because the logical view of the database can be implemented independently of the physical view of the database system.

Although the relational view of data does have many advantages, problems still do exist. These problems are realized when considering the efficiency of the actual implementation in terms of storage management and processing time. Thus, when the AFIT community became interested in installing a relational database system for teaching and research purposes, concern was expressed in this area. The original design of the AFIT relational database system takes into account these inefficiences and

considers methods for increasing the efficiency of a relational database system (Ref 16). The actual implementation includes optimization software. In addition, the access structure suggested has the potential to speed up processing time. During this thesis effort, these considerations still need to be addressed and implemented in order to determine whether or not the potential to decrease processing time and increase the efficient use of memory space can be realized.

STATEMENT OF THE PROBLEM

The purpose of this thesis investigation is to further the implementation of the AFIT relational database system, taking into consideration the design of algorithms and indexing techniques which have the potential to reduce processing time and introduce efficient memory usage. In addition, the further design and implementation of the editor and low level access structure is to be accomplished.

SCOPE

The scope of this thesis effort is to investigate techniques and algorithms for implementing file structures and relational database operators. At the beginning of the thesis effort the scope also included the design and implementation of the edit modules, the low level access structure, and the run modules.

GENERAL APPROACH

The general approach began with a literature review.

The goals of this literature review were as follows: 1) One goal was to review the research from which the AFIT relational database system originates. This review of the literature included a review of query optimization techniques and file structure implementation. 2) The second goal was to understand what had been actually implemented. Not only was it important to understand the actual implementation of this database system, but in addition, it was necessary to understand alternative approaches to the design and implementation of other relational databases for minicomputer systems. 3) The next goal was to become familiar with the hardware and software facilities available. 4) The final step consisted of designing and implementing the edit modules taking into consideration the need to make these modules more user oriented. In addition, the low level access mechanism was designed and partially implemented. The design and implementation of these modules were approached in a top down modular fashion so that the sequence of processing could be determined.

SEQUENCE OF PRESENTATION

The sequence of presentation is as follows: The second chapter discusses the original design and implementation decisions concerning this database system. A presentation of a query optimization technique is given. In addition, the access structure recommended by Roth is presented.

Chapter Three presents the theoretical aspects of this thesis effort. Specifically, the data manipulation language designed is discussed. Also, the access structure is described in detail. In Chapter Four, the actual design and implementation decisions concerning the editor and the low level access structure are discussed. Chapter Five presents the techniques used for system validation. The final chapter, Chapter Six presents the conclusions drawn at the end of this thesis effort.

II. BACKGROUND

INTRODUCTION

Much has been written concerning various theoretical aspects of the relational view of data since E. F. Codd first formalized this concept in 1970. However, the database community has not only addressed the relational database concept from a theoretical viewpoint, but a considerable amount of work has placed emphasis on the actual implementation of a relational database model for various computer systems. Currently, special interest is shown for the implementation on minicomputer systems, and to this date several implementations do exist. Such implementations include the RISS relational database (Ref 11), INGRES (Ref 7), and the DBMS-II system (Ref 8:16). Various AFIT faculty members and students have also expressed an interest concerning the implementation of a relational database for a small computer system. In 1979, 2nd Lt. Mark A. Roth designed and partially implemented such a system (Ref 16).

The design and implementation of this relational database system is of special interest because it addresses several key aspects of relational database design. Specifically, Mark Roth's final design and implementation provides a means for data definition and for data manipulation. The state of the AFIT relational database system at the conclusion of Mark Roth's thesis effort is

shown in Figure 1. The original system was designed and partially implemented on the Intel Series II (Ref 16). However, implementation was continued by Mau on the LSI-II using UCSD Pascal (Ref 9). The following sections discuss the modules which constitute the major portion of the database system.

THE DATA DEFINITION FACILITY

The data definition facility provides a means for defining domain and relation definitions. This facility was implemented in a previous thesis effort (Ref 16). However, several implementation decisions were questioned. Specifically, the original implementation of the DDL facility allows any user to have access to this facility. This implies that the user can not only create domain and relation definitions, but the user also is given the power to destroy domain and relation definitions. As a consequence, the DDL facility was removed from the existing program as a part of this thesis effort. The DDL facility now can only be invoked by the database administrator. It seemed appropriate to entrust to the DBA the duty and the power to resolve the policy decisions which must be made concerning domain and relation definitions. Specifically, it was felt that the DBA should have centralized control of the domain and relation definitions so as to aid in maintaining the security and the integrity of the database system. The resulting DDL facility is shown in Figure 2.

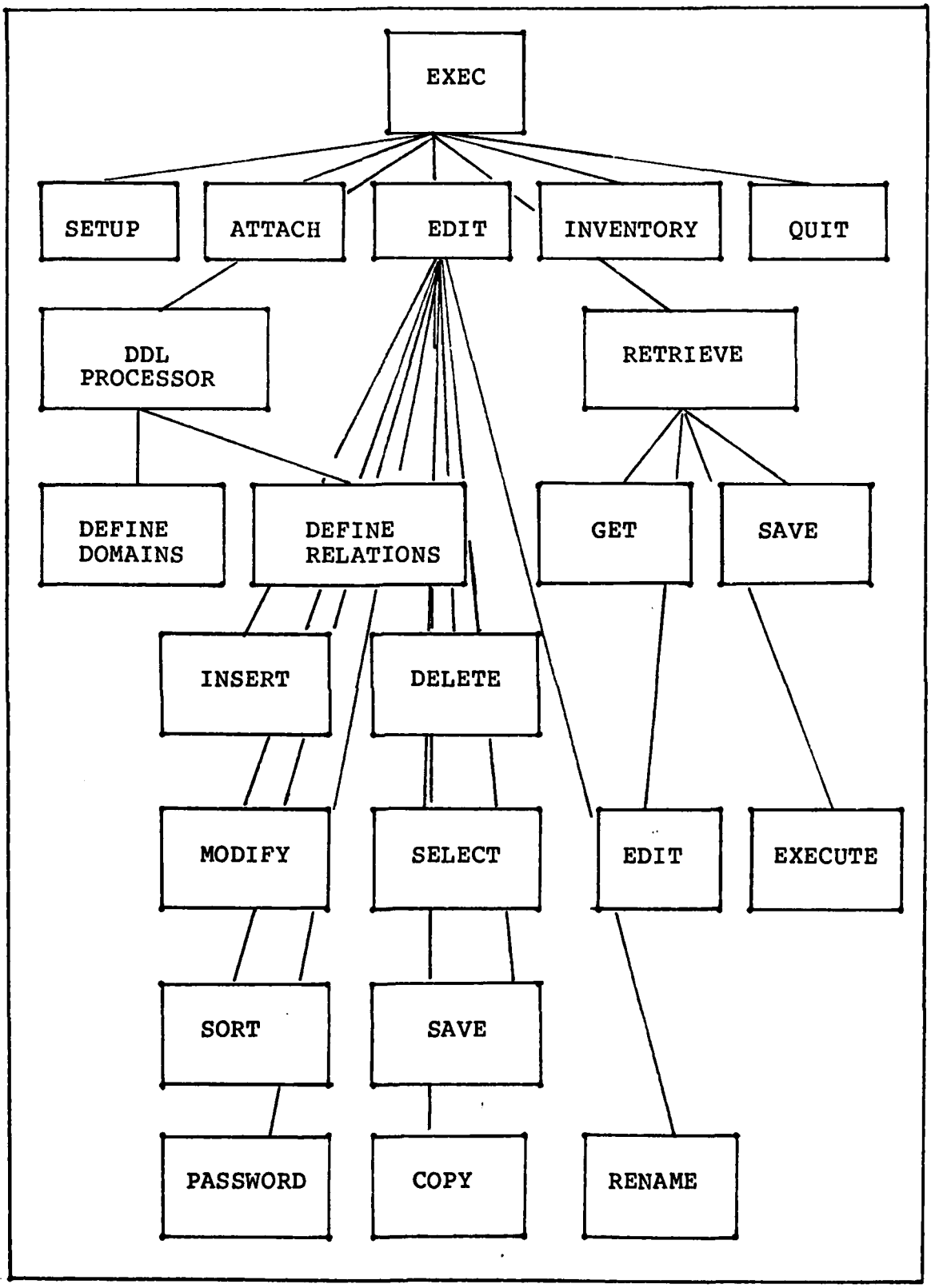


Figure 1. Roth's Database System

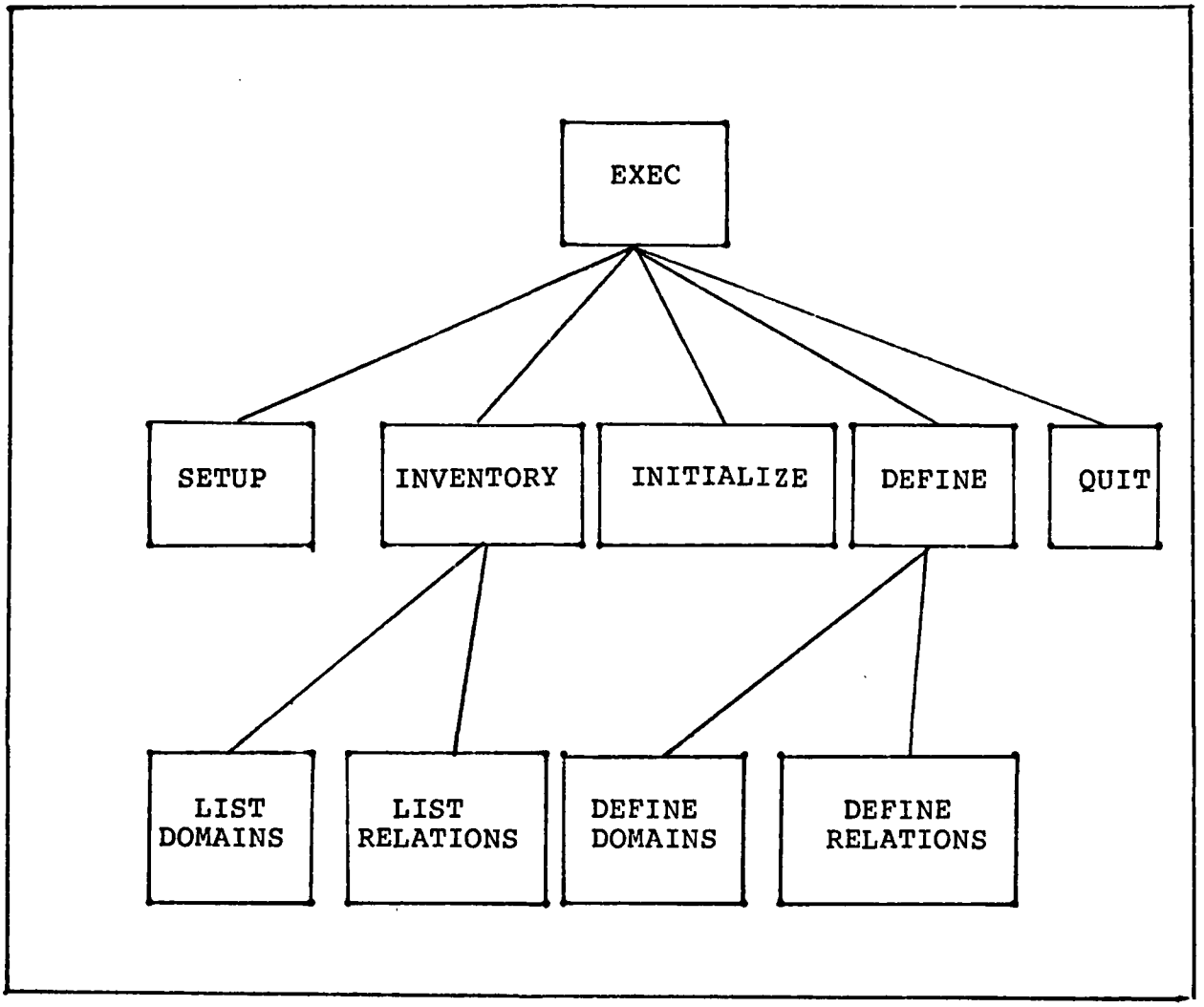


Figure 2. The DDL Processor

THE DATA MANIPULATION FACILITY

Although it was not fully implemented, the data manipulation facility (DML) will provide the user with a means for querying the database. As shown in Figure 1, it consists of the retrieve module which allows the user to enter a number of queries into a command file. The user has the option of saving this command file for later use. Also, an execute module exists which allows the execution of this query command file. During the execution of the command file, the queries are optimized by a technique formulated by Miles Smith and Philip Chang (Ref 18). This optimization technique is discussed in the following section:

Query Optimization. In the past, optimization studies in relational systems have concentrated on low level efficiency such as optimizing the access paths to information. However, by performing certain transformations on a query, a means of high level optimization can be provided. By providing the means for high level optimization, a gradual refinement of the query process can be obtained so that the work expended by the low level optimization process or access structure is essentially minimized. In order to realize the full potential of such an optimization process, all queries are relational algebra expressions. Since relational algebra treats and manipulates whole relations as single entities it offers more scope for high level optimization as

compared to that of the relational calculus (Ref 18: 569).

The Query Optimizer. The optimization technique has the following organization. It consists of the syntax analyzer, the tree transformer, and the coordinating operator constructor (Figure 3). The tree transformer and the coordinating operator constructor perform the major portion of the optimization process. They are described in the following paragraphs.

The Tree Transformer. The tree transformer performs two major functions. The first function is to take the output of the syntax analyzer, which is a query tree, and perform correctness preserving transformations on the tree. This essentially involves moving selects and projects down the operator tree. This is especially advantageous if directories exist on the specified attributes which constitute the select or project criteria. If directories exist on the specified attributes, the resulting tuples can be accessed in a random fashion. This has the potential to speed up processing time. An additional advantage obtained by moving selects and projects down the operator tree is that together they decrease the net area of the relations passed to higher operators. Specifically, moving selects down may introduce a reduction in the number of tuples to be processed by future operations. Moving projects down serves to decrease the width of the resulting temporary relations. In addition, the number of tuples may be

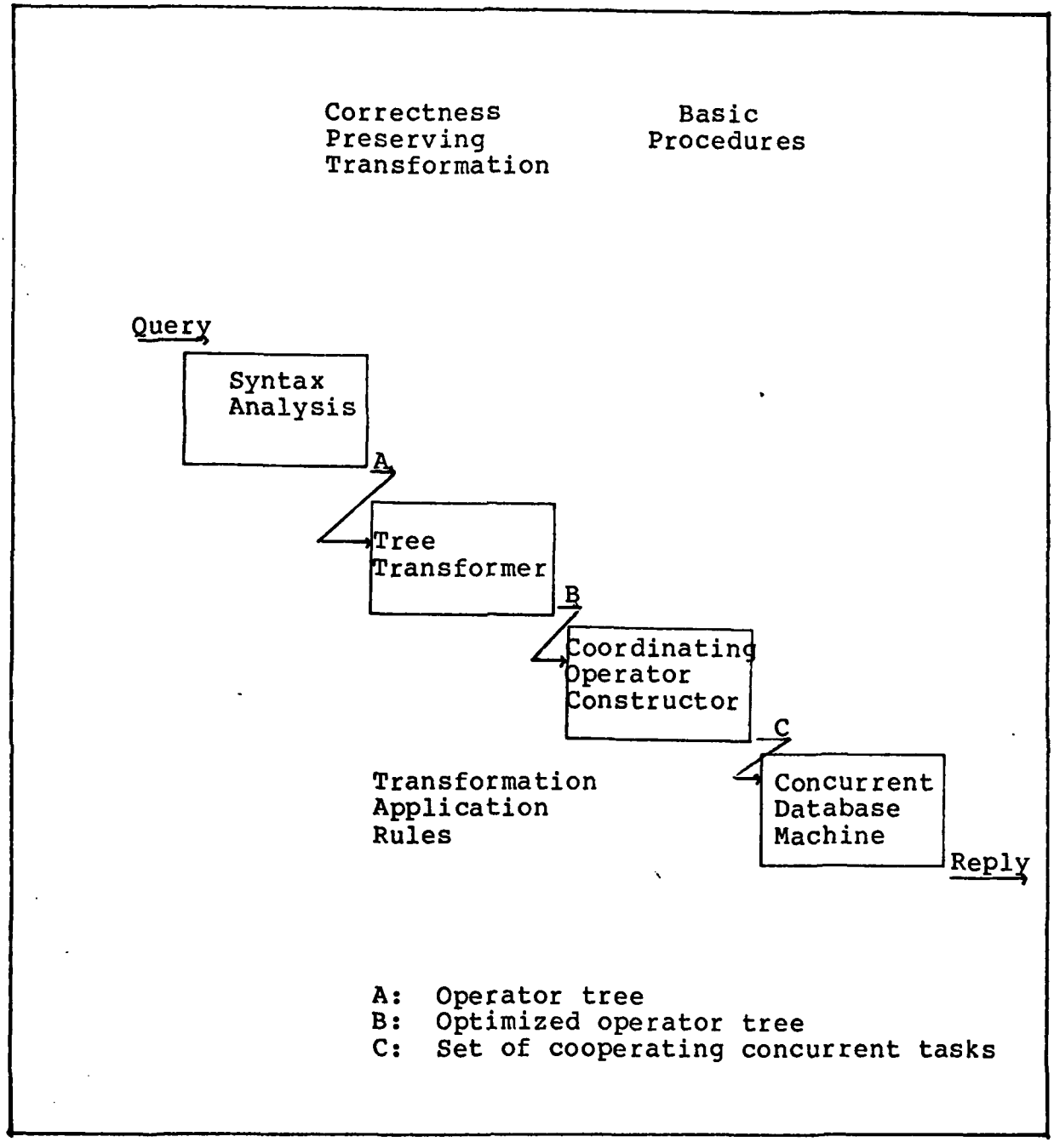


Figure 3. The basic organization of the query optimizer.

decreased by the elimination of duplicates (Ref 16: 570).

The second function of the tree transformer is to replace a subtree of set operators on the select of a common relation by a compound Boolean operation (Ref 18: 573). The advantage of this operation is that it ensures that the common relation is never read more than once. This function also saves processing time.

Together, the two functions of the tree transformer serve to decrease the amount of memory or disk space used to store temporary results. In addition, they help decrease the amount of processing time otherwise performed on redundant information.

The Coordinating Operator Constructor. The next phase of the optimization process is performed by the coordinating operator constructor. The function of the coordinating operator constructor is to take the transformed tree and implement each operator represented from a set of basic procedures in such a way that the sort orders of the intermediate relations are "optimally" coordinated. The advantages of performing this optimization process are twofold. One advantage is it speeds up search time if directories exist for the domains chosen as the sort order for the resulting relation. Two, if the sort order chosen happens to be a primary key, a relatively fast procedure can be used to implement the operation. This is because the operation does not have to eliminate duplicate tuples. Thus processing time is

comparatively reduced.

The general procedure for the coordinating operator constructor consists of two passes of the operator tree. On the upward pass, each branch of the tree is labeled with a set of sort orders which are provided from lower operations. On the downward pass, the set of preferred sort orders are examined and a final choice is made as to what preferred sort orders will be implemented (Ref 18: 573-576).

Mark Roth's Optimization Process.

The optimization process implemented by Mark Roth is very similar to the optimization process just described. However, he does include some additional features. The basic query optimizer, as defined by Smith and Chang, only attempts to optimize single expressions which result in a single relation. However, Roth expanded this optimization process to include the optimization of multiple queries where several relations are the result. This expansion introduces opportunities to exploit shared subtrees not only within a query, but among different queries, and to optimize the execution order of various queries (Ref 16: 55). The variations made to the optimization process are concentrated in two modules. One is the tree module, which outputs a network of shared trees. The other is the splitup module, which takes this network of trees and outputs an ordered, optimized forest of separate trees in which all shared subtrees have been removed. The remainder

of the optimization process performs the functions of the tree transformer and the coordinating operator constructor as described by Smith and Chang on the forest of trees (Ref 18: 52- 78). However, the coordinating operator was not fully implemented by Roth's thesis effort.

SUMMARY

The implementation of the AFIT relational database by Mark Roth addresses several key aspects related to relational database design. The system consists of a DDL facility which gives the database administrator complete control to define domain and relation definitions. In addition, the system consists of a data manipulation facility which will allow the user to build a query command file. This command packet is then optimized by the technique previously described. Suggestions were made for the inclusion of an editor which will allow the user to perform various editing functions on a relation. Once these modules were implemented, the design and implementation of the low level access structure was considered.

ACCESS STRUCTURE CONSIDERATIONS

It is assumed that the amount of information to be stored in a database system is too large to be stored in primary memory. Thus, secondary memory provides the storage capability necessary to handle large amounts of information. As a consequence, an important aspect of database design is to define the associated mapping

between the secondary storage structure and the database management system (Figure 4). This mapping or access mechanism provides the means for locating information that resides in secondary storage so that the data can be brought into main memory when needed and converted into its internal representation.

Factors in Access Path Determination

The performance of a database system in accessing its data depends largely upon how well the organization of the data is suited to the types of access that are requested (Ref 20: 66). No single database organization can be considered best for retrieving data under all circumstances. For example, a sequential file organization works well for sequential processing, but tree or hashing structures are better for randomly accessed data. Also, logical relationships may exist between the data. Since it is often the case that certain data is retrieved by specifying these logical relationships, a physical organization tailored to suit these logical relationships is desirable. Hence, three factors which affect access path determination and in turn database performance are the type of access, or query, that the user requests, the logical relationships defined on the data, and the physical organization of the data.

In essence, an access path into a database is a series of steps which must be taken in order to search the database and retrieve the data requested by the user.

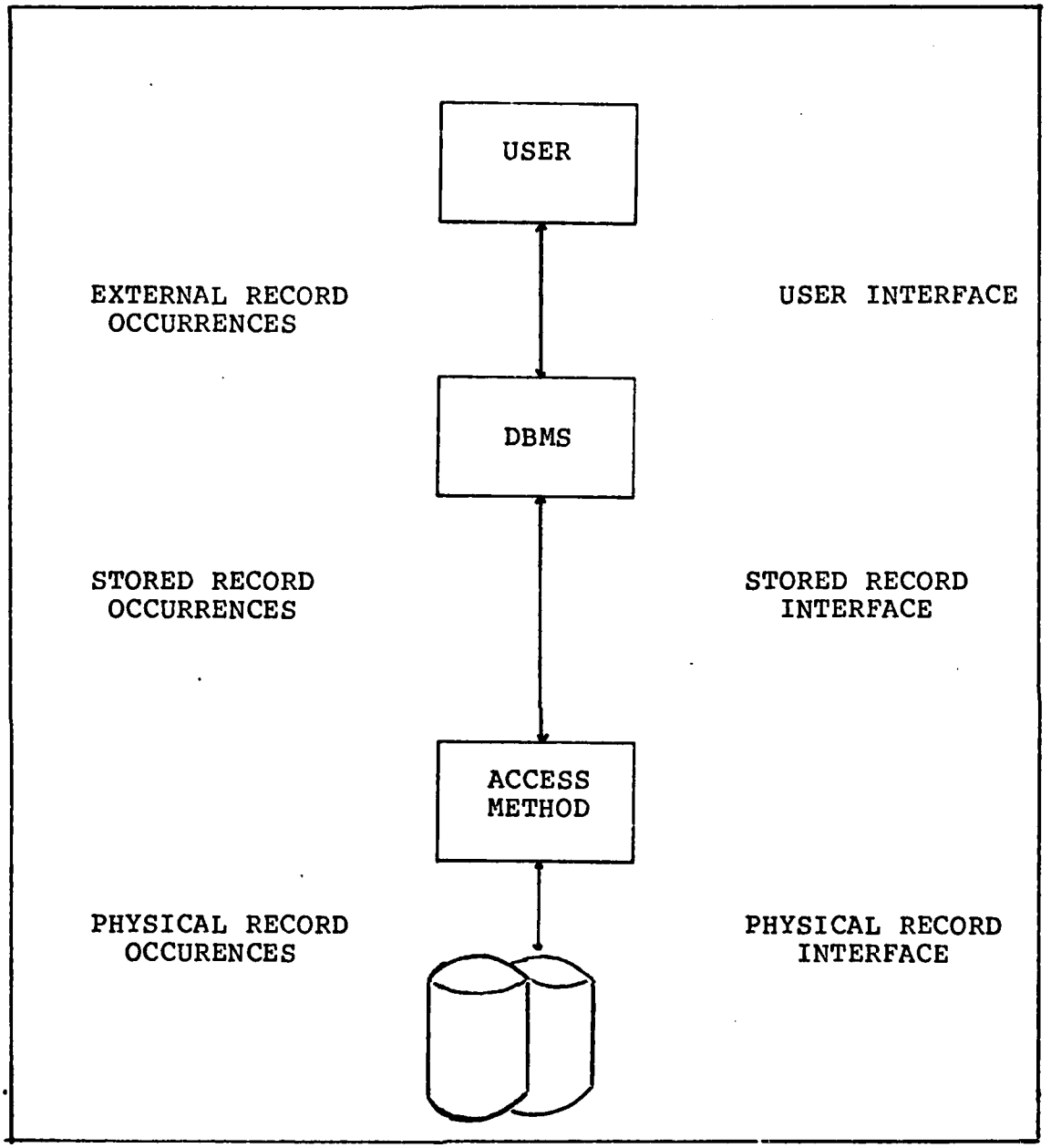


Figure 4. Interfaces in a DBMS System

These steps may include such operations as searching indexes, tracing linked lists, or sequentially scanning the data for the requested information. A goal of any database design is to provide an access path or access paths which accomplishes these steps and retrieves data efficiently. This goal has been the focus of past optimization studies and hence, the three performance factors mentioned have been formally studied (Ref 20). However, because there are several factors involved, the determination of an access path design is still a relatively complex process. In addition, the determination of an associated indexing technique, if considered, may also be complex.

B-Trees

Today, several file organizations along with their associated indexing techniques exist. However, the B-tree data structure formalized by Bayer and McCreight (Ref 11) has proven to be an effective method of organizing an external file when the operations of searching, insertion and deletion must be supported. The reason is that performing these operations on a Btree takes at most time logarithmic in the size of the file (Ref 15: 174).

Definition of a B-tree. A formal definition of a B-tree is given as follows: A B-tree, T, of order M is an M-way search tree that is either empty or is of height >= 1 and satisfies the following properties (Ref 6: 499):

(i) the root node has at least two children

(ii) all nodes other than the root node and failure nodes which are the terminal nodes have at least M/2 children

(iii) all failure nodes are at the same level

An example of a Btree is given in Figure 5.

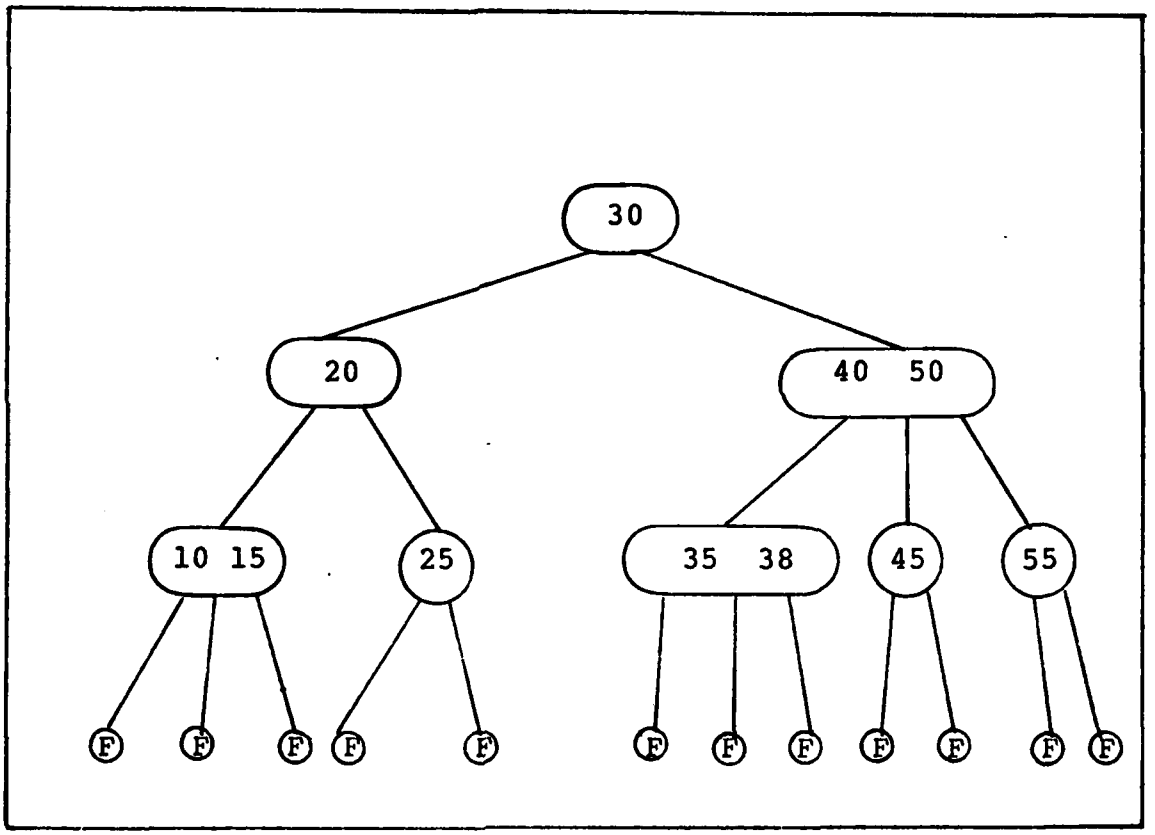


Figure 5. An example Btree

Advantages and Disadvantages of B-trees

Several advantages of the B-tree structure exist. The major advantage of this structure is that the tree can be kept balanced during insertions and deletions with a known worst case update cost. Therefore, a small worst case search time is always guaranteed. However, an additional means of measuring the performance of a B-tree indexing technique exists, and that is in terms of space utilization. Not surprising, however, is the fact that certain tradeoffs do exist between space optimal B-trees and time optimal B-trees. As a result of a study on this topic by A. L. Rosenberg and L. Snyder, it was found that space optimal trees are nearly time optimal, but time optimal trees are nearly space "pessimal" (Ref 15). Thus, the tradeoff is strongly in favor of space minimizing B-trees. This fact can be an important influence on the design of an accessing technique on a minicomputer system.

Although the B-tree structure does seem to have its advantages, it does have its disadvantages. As previously stated, there is no single database organization which can be considered best for retrieving data under all usage circumstances. Unfortunately, this is the case with the B-tree. Specifically, the B-tree may not perform well in a sequential processing environment (Ref 1: 128). Thus, other alternative B- tree structures can be used.

B-tree Variants

Several variations of the B-tree structure exist (Ref

5). One such variation is the B+tree in which all keys reside in the terminal nodes. The upper levels, which are organized as a B- tree, consist only of an index. The leaf nodes are usually linked together left-to-right. The linked list of leaves is referred to as the sequence set. Sequence set links allow easy sequential processing.

Another variation of the B-tree is the B*tree. The B*tree is defined to be a B-tree in which each node is at least 2/3 full instead of just 1/2 full. Consequently, the structure guarantees that storage utilization is at least 66% (Ref 1: 129). Also, the B*tree has the potential to decrease search time since the height of the resulting tree is smaller than the height of a B-tree with the same number of keys.

The Access Path Structure Recommended

The index structure suggested by Mark Roth is a variation of the Btree structure described. Specifically, the structure chosen was formalized by Theo Haerder and is called the generalized access path structure (Ref 3). In addition to the advantages mentioned above, this structure has the power to limit data redundancy. The theoretical background of this structure will be explained in detail in the following chapter.

THE EXISTING RUN MODULE

The run modules are those modules which perform the function of the coordinating operator constructor. In addition, these modules should actually retrieve the

appropriate relations from disk and perform the operation specified by the optimized tree. At the beginning of this thesis effort, the run module had been partially implemented. However, it could not be completed without the use of the low level access structure or the implementation of the relational operators. The code that was completed by Peter Raeth (Ref 14) performs part of the function of the coordinating operator constructor as descibed by Smith and Chang (Ref 18). Specifically, the code implemented makes 3 passes over the operator tree in order to determine the preferred sort orders. The first pass is an upward traversal of the operator tree in which the domain fields resulting from each node's operator are attached to the respective node. In addition, the sort orders of stored relations are attached to the nodes where these relations are used as input. On the second pass, which is also an upward pass of the tree, the domain fields are used to label each node with the set of preferred sort orders which can be efficiently generated from lower operations. This is accomplished by using the "up" rules for choosing preferred sort orders (Ref 18). The third pass is a downward traversal of the operator tree. During this final pass, the sort orders which can be most efficiently supplied from below to a given node and what sort order the current node must pass up are known. Thus, a preferred sort order is chosen from below and is now available in order to construct an appropriate

implementation of the current operator (Ref 18).

Although the run modules, as implemented by Raeth, did make available the proper preferred sort order, they still were not complete. Specifically, Codd's relational algebra operators still needed to be implemented from a set of basic procedures (Ref 18: 577- 579). These basic procedures are used to implement several variations of an operator depending on varying factors such as whether or not directories exist for a domain or whether or not a relation is sorted on its primary key. Although several operators can be recognized in a command packet, the actual run modules only recognize the equijoin, project, select and intersect operators.

SUMMARY

At the beginning of this thesis effort, it was apparent that a great deal of effort was expended in order to implement the AFIT Relational Database System which provides a means for data definition and data manipulation. It was the intention of earlier work to implement a system that would run efficiently on a microcomputer system. This is the reason for the optimization software and the suggestion to use the generalized access path structure to implement the low level access mechanism. The remainder of this thesis addresses these concerns. Specifically, the theoretical aspects surrounding this investigation are discussed in the following chapter and the design and implementation is

described in Chapter Four.

III. THEORETICAL CONSIDERATIONS

INTRODUCTION

This chapter describes the work that was done prior to the initial design and implementation of the editor and access path mechanism. Through meetings with those concerned with the implementation of the AFIT database system and through referencing several articles and books related to the design of relational database systems, various ideas were formulated for the initial design of the editor and the access path mechanism. Also, ideas for future implementation considerations were developed.

In order to provide some guidance for the design of the edit modules, it was necessary to investigate the suggestions made concerning the data manipulation language formulated and the edit module design recommended. By understanding the reasons for the choice of DML, an understanding of how to change the DML, if necessary, could be achieved so as to improve the design of the editor and in turn improve program maintainability and "user friendliness". This investigation is discussed in the first section.

In order to actually manipulate the data in the system, the access path mechanism needed to be implemented. The access path structure selected was that formulated by Theo Haerder and is called the generalized access path structure (Ref 3). This access mechanism is

discussed in the second section of this chapter.

Once these areas had been investigated, an understanding was obtained about what the functions of these modules should be and what would be suitable methods and algorithms that could be designed and used to implement these modules in order to create a more maintainable system.

FEASABILITY OF THE DATA MANIPULATION LANGUAGE DESIGNED

In this section, a discussion of the data manipulation language originally chosen and the reasons for choosing such a language are given. Then, the concept of extending the DML to perform edit operations by using the relational algebraic operators is presented.

The Language Chosen

The type of language specified by Mark Roth to be the basis for the design of the data manipulation language was the relational algebra. The reasons for choosing an algebra-based language was that it was considered easier to learn, easier to use, and easier to implement than a relational calculus based language. In addition, an algebra based language provides all the operations necessary to manipulate the data with the inclusion of the insert, delete, and modify operators (Ref 16: 21).

Agreement with the DML Chosen

After determining the class of potential users of this database system to be individuals who were familiar with programming techniques, it was decided that the choice

of a formal algebraic language was feasible. In addition, it was felt that this type of language could be "easily" learned. The algebraic languages are basically procedural in nature and for this class of user, the process of specifying a sequence of algebraic operations in order to retrieve information should be a more natural process than using a calculus based language which tends to be an uncomfortable language to learn because of the quantifiers and bound variables which are inherent in this language (Ref 16:19). In addition, it was felt that this choice of language would not stop the user's ability to interact with the database system because most users would be students who are capable of interacting with computers in this manner. In addition, if an alternative means of interfacing with the retrieval portion of the database were implemented, a formal language interface would still need to be implemented to allow batch processing capabilities. Also, by defining a formal language, unnecessary interactions with the computer could be minimized because the user would not have to be continously prompted for information. Instead all information would be supplied by the user in one formatted command.

EXTENSION OF THE DML

In addition to these considerations, an interesting question concerning the extension of the data manipulation language arose. Specifically, Mark Roth's approach was to consider the data manipulation language to consist of two

separate sublanguages. Particularly, he considered the DML to consist of one language that would allow the user to perform edit functions and another language that would allow the user to perform retrieval functions. However, it was suggested that the data manipulation language proposed for processing retrieval operations could also be used for processing editing operations. Specifically, the algebra based language proposed by Roth includes all the traditional relational operations as discussed in the definition of a relational algebraic language. These include the union, intersection, projection, selection, join, division, and difference operation. In addition, Roth included statements for the insertion, deletion, and modification of the tuples in a relation. However, the question still remains as to whether or not the insertion, deletion, and modification operations can be implemented by the traditional relational operations.

Roth's inclusion of separate statements for the insertion, deletion and modification of tuples is not unusual. For example, several relational algebra based languages such as the System R data manipulation language and Query By Example (QBE) have specifically defined statements for performing insertions, deletions and modifications. This is not surprising, since the relational algebra is basically considered a retrieval language (Ref 2: 213), and thus additional operators for update operations are added. However, the possibility

still remains that the traditional algebraic operators can perform the job of these update operators.

In Roth's implementation of the relational database system, it may prove beneficial to consider the relational algebra as both an update and a retrieval language. If so, separate procedures would not have to be designed and implemented to handle both update and retrieval operations. Instead the data manipulation language defined could be used to perform update and retrieval operations and the optimizer could be used to optimize a query packet which contained operators which would have the effect of updating relations and retrieving information. Another approach would be to define separate update statements for the insertion, deletion and modification of relations. These statements could be analyzed and the insert, delete or modify operator could be translated into the corresponding set of relational algebra operators which would produce the equivalent results.

Although this does seem to be a reasonable alternative for the design of a relational algebra based language, some aspects still need to be considered. The union operator may not be a satisfactory substitute for the insert operator. The reason is that the union operation may not reject an attempt to "insert" a tuple that is a duplicate of one that already exists (Ref 2: 213). A solution could be to require that the implementation of each relational algebraic operation eliminate duplicates

before producing a result. The actual design of the edit module is discussed in more detail in the Chapter Four.

THEO HAERDER'S GENERALIZED ACCESS PATH STRUCTURE

The initial design phase of this relational database system included investigating various methods of improving system efficiency and user responsiveness. As a result of this research, it was found that the performance of the access mechanism implemented could greatly affect the overall performance of the entire data base system. Specifically, an access mechanism which does not provide the capability for fast associative and sequential access and for fast navigation from one tuple to another related tuple would not prove to be responsive to user needs in a current information processing environment. To explain further, current database systems should provide a means for the user to request and quickly access information and receive a response in a relatively short period of time. If the access method only provides access to tuples by using key comparison techniques such as sequential access to information in the data base system, processing time could prove to be very long for rather complex queries. As a result, the system would be considered slow to respond to user requests.

In order to improve system responsiveness in the past, two separate access path mechanisms were often implemented to allow associative and sequential access and navigation from one tuple to another. Secondary indexes

were implemented to provide associative and sequential access while pointer chains were implemented to provide navigation from one tuple to another. Although the requirements for associative, sequential and navigational access were met, a disadvantage did exist by implementing two separate access path mechanisms. Two different sets of procedures needed to be implemented to support the different access paths.

Because Theo Haerder's access structure (Ref 3) does not require the implementation of two sets of procedures to support its implementation, and because it does provide for associative, sequential, and navigational access, it was felt that Roth's recommendation (Ref 16) for the use of such an access mechanism could be justified. As a result, a detailed review of Theo Haerder's access path structure was necessary. In the following paragraphs a presentation of Haerder's access mechanism is given.

In order to present Theo Haeder's access path mechanism, the terms link and image will be explained for they are key concepts in the discussion of the implementation of the generalized access path mechanism. The sequence of presentation will be the following: First, these two concepts will be presented in the form of definitions. Next, the implementation technique for images and links will be discussed. Finally, the combined access path structure and the generalization of this access path structure will be presented.

Definitions.

It is important to provide sequential and associative access paths to information so as to decrease the time to process such requests without performing time consuming sorting routines or search operations. An access path which provides value ordering and associative access by one or more attributes to one relation is called an "image". A formal definition is given in the following paragraph:

Definition of an Image. Let R be a relation with attributes A1,...,An. An image of the attribute Ai of R, $i \in \{1,...,n\}$, is a mapping from values in Ai to those tuples in R which have that value for the ith attribute. Additionally, these sets of tuples qualified by values of Ai are ordered according to the sorted sequence of values of Ai (Ref 18: 287);

An access path providing navigational access from one tuple of one relation to another tuple in another relation is called a "binary link".

A more formal definition for a link is presented as follows:

Definition of a Link. Let R be a relation with attributes A1,...,An, S be a relation with attributes B1,...,Bm; F(Ai) = F(Bk) for the domains F(Ai), F(Bk), $i \in \{1,...,n\}$, $k \in \{1,...,m\}$; Let Ai be a candidate key of R. The link between R and S with regard to Ai, Bk is defined as the set L(R(Ai), S(B(k)) := $\{(r,s) | r \in R, s \in S, pr(r) = pr(s)\}$, where pr (r) and pr (s) are the projections to the

components of r and s which correspond to attributes Ai and Bk, respectively.

These two definitions are presented so as to facilitate the understanding of the information presented in the following paragraphs:

Implementation Technique for Images.

In the past, images and links were implemented and maintained in separate structures. An image was implemented by using a multipage index structure which contained pointers to the tuples called tuple identifiers (TID's). The pages of this index were often organized as B*trees. For nonleaf nodes of this B*tree, an entry consisted of values of single or compound attributes and a pointer which addressed another nonleaf page or a leaf page in the same structure.

For the leaf nodes, an entry consists of a combination of key values. Along with these key values are a variable length list of TID's for tuples having the key values listed. This variable length list is sorted in ascending order. In order to identify the length of the TID list, an additional field which contains length information is stored with each key. In addition, the leaf pages are chained together in a doubly linked list so that sequential access can be supported. If the total storage space is used for a leaf page, overflow pages may be used to keep the overflowing information of a leaf page. An example of an image on the attribute CITY for the Flight relation is given in Figure 6.

Implementation Technique for Links.

A binary link provides navigational access by connecting tuples in one or two relations on matching attribute values. Usually, binary links are implemented by using chaining techniques with TID's or physical pointers.

For example, links are represented in the Relational Storage System by storing the TID's or the next, prior, and owner tuples in the prefix of the child tuples and by storing at least the TID of the first child tuple in the parent tuple as shown in Figure 7. In this example, one tuple of the owner relation Class(C#) is linked to n tuples

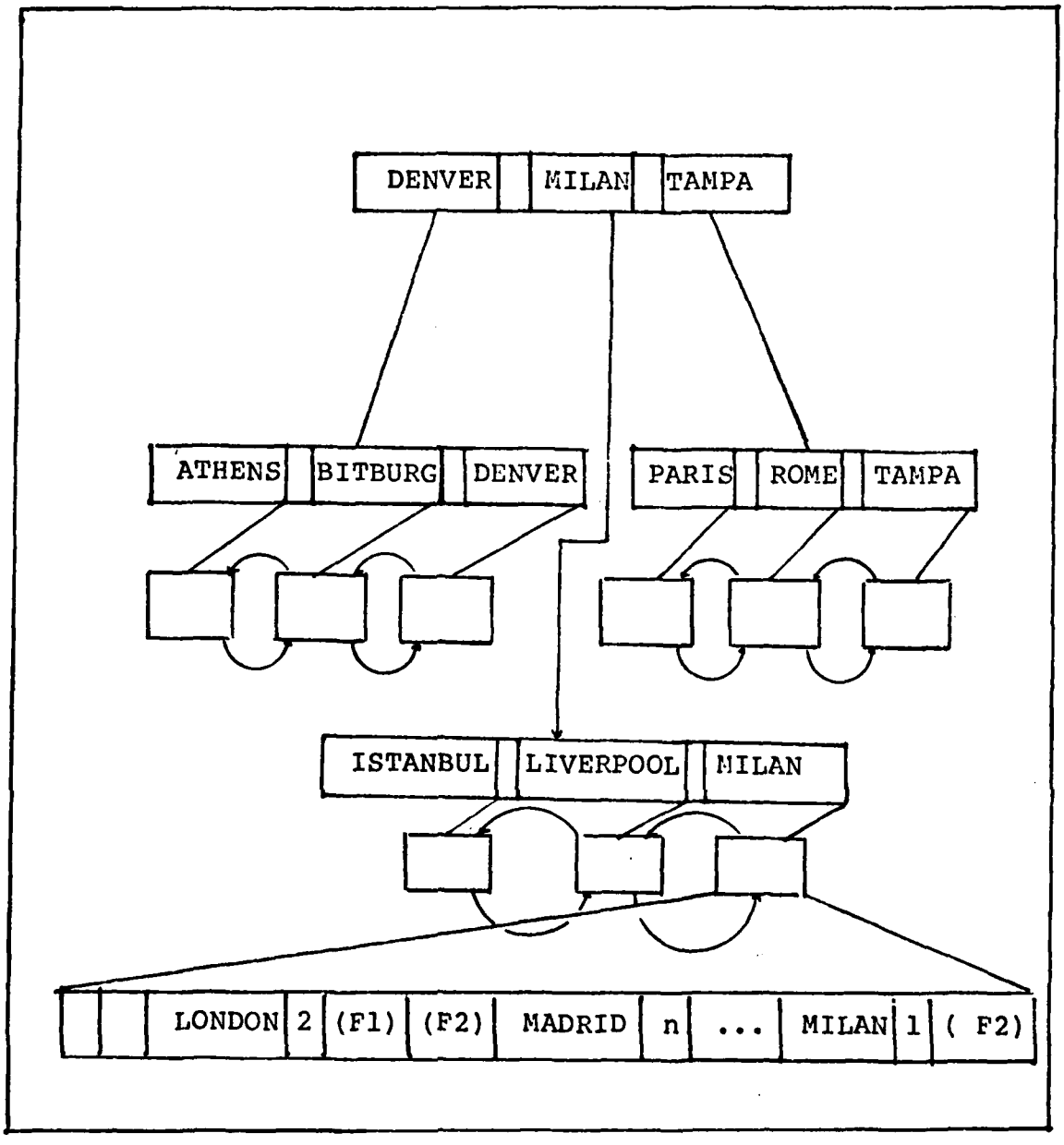


Figure 6. An Image on the attribute CITY

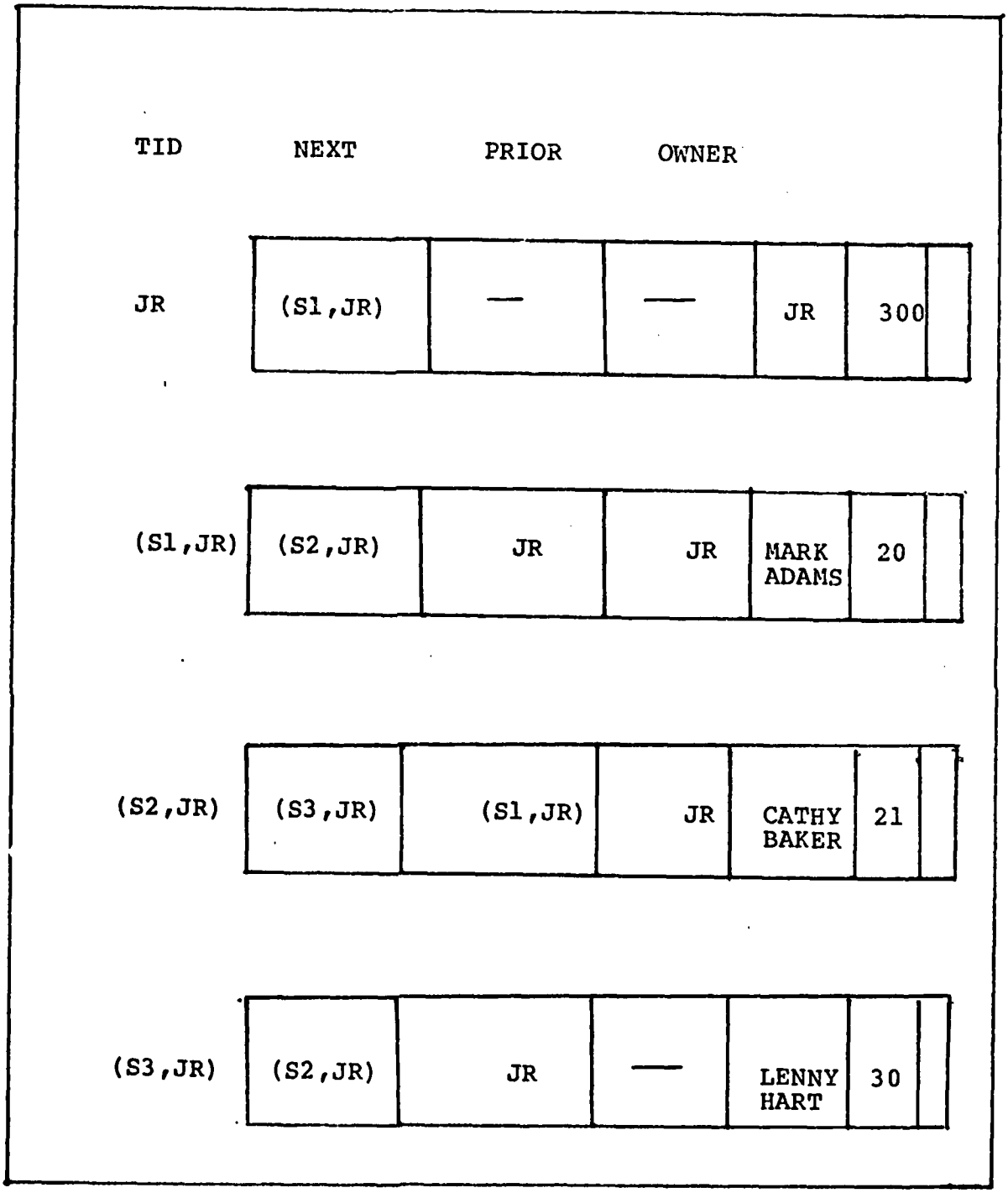


Figure 7. Link Implementation L(Class(C),Student(C))

of the Student(S#,C#,...).

Implementation Technique for a Combined Access Path Structure.

A binary link provides a direct path from single tuple (parents) in one relation to sequences of tuples (children) in another relation based on an attribute value in the parent relation. The advantage of using a link compared to that of using an image to find child tuples is that a link provides the direct access to a tuple whereas the use of an image may involve a complete traversal of a B*tree structure. The reason is that an image is an index for an attribute of one relation. Thus to find a child of another relation, an additional image must be searched. As a result, several additional page accesses may be involved in order to find the child or parent tuple. Consequently, this may be a very time consuming process and the goal of providing fast navigational access may not be achieved.

An important characteristic of the relationship between tuples of one or different relations should be pointed out. That is, these relationships are expressed explicitly by attribute values in the relational model and this key property allows combined images on the same domain to serve also as link structures. In other words, a B*tree structure can represent an image for several relations as long as the attributes of the various relations are defined over the same domain. Thus the advantage of image and link access can be combined into one structure. This is accomplished by using a different organization for the leaf nodes of the B*tree. The nonleaf nodes look exactly as in the single image implementation. In the leaf node, separate TID lists for the various relations, together with the related length information fields, are stored for each key. The lists for the parent relation contain only one TID entry, while each variable length list for the child relation contains the sequence of TID's for the children related to a particular parent tuple. In Figure 8, an example for the STUDENT and CLASS relations is given.

With this access path structure, the traversal of an additional B*tree structure can be avoided when the child tuples are to be accessed after the parent tuple is located. In either case, it must be assumed that the owner tuple is found via an image access I(STUDENT(S#)). If the leaf page containing the required key for the tuple of the owner relation is in primary memory,

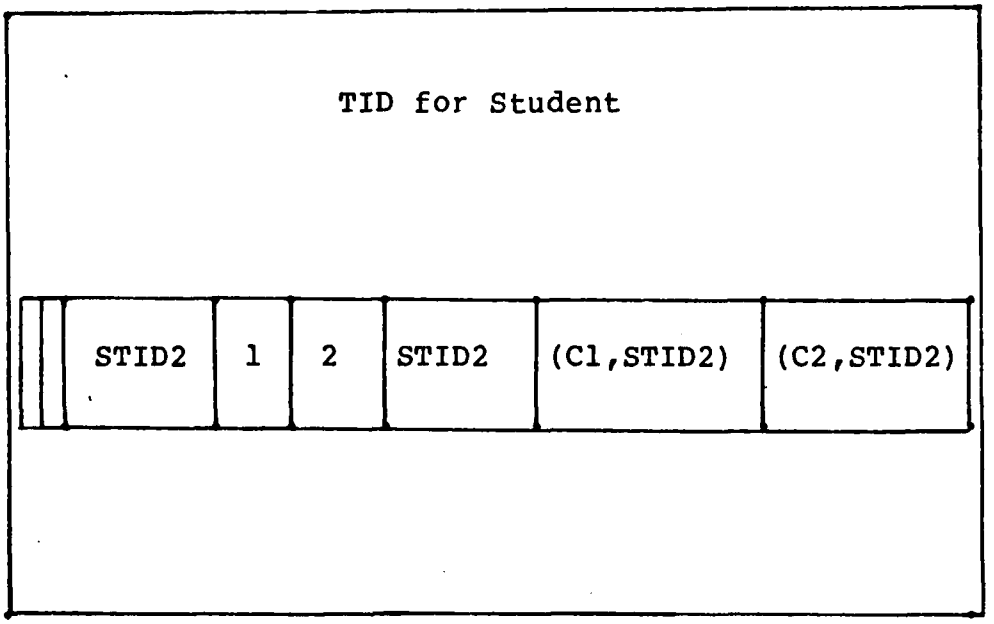


Figure 8. Combined Implementation of Link L(Student(STID), Class(STID) and the Images I(Student(STID) and I(Class(STID))

then the subsequent navigational accesses to the tuple of the member relation are at least as fast as the accesses via the binary link.

Generalization of the Combined Access Path Structure

The combined access path structure replaces the separate image and binary link access paths by joining the various characteristics of both into one structure. However, this combined structure can be extended so that it becomes what Theo Haeder refers to as the "generalized access path structure". The implementation technique used is presented by the following statements: All variable length TID lists belonging to the various attributes in different relations which are defined on the same domain are stored with their related domain value. The format of the nonleaf pages is the same as for the image and combined access path. However, the leaf pages contain for each key up to m variable length TID lists together with m length information fields. If an actual domain value is not defined for an attribute, then the corresponding TID list does not exist. In addition, a zero entry is indicated

in the corresponding length information field.

SUMMARY

The generalized access path structure has several features which suggest that it would be an efficient access mechanism to implement. For instance, instead of supporting specialized modules for each of the access path types, only one unified set of modules working on the combined access structure is necessary.

An additional advantage is that the generalized access path structure supports in a natural way the join of relations, because the access information for tuples of different relations having matching values is stored close together. Not only does the access path structure support the implementation of joins, but the structure easily supports many-to-many joins (Ref 3: 294).

Another advantage of this access path structure is that some queries can be answered without looking at the actual data. For example, there is no need to add an attribute , Total (number of members in a class), to the Class relation because the values of this attribute can be derived from a generalized access path without additional costs (Ref 3 : 294).

As can be seen, this access path mechanism has the potential to save a considerable amount of processing time as compared to implementing separate access paths for images and links. The actual implementation concerns are discussed in Chapter IV.

IV. SYSTEM DESIGN AND IMPLEMENTATION

INTRODUCTION

This chapter describes the work that was accomplished during the actual design and implementation of the Data Definition Facility, the Editor, and the low level access structure.

Prior to the design of these modules, a considerable amount of research concerning the implementation of the AFIT Relational Database System was accomplished by Mark Roth (Ref 16). The findings of this research had the most effect on the design and implementation of the Data Definition Facility and the editor. Specifically, Mark Roth's documentation of his thesis served as a baseline for the further design and implementation of these modules and its function as a guideline will be discussed in parallel with the design and implementation decisions actually made throughout this thesis effort.

In this chapter, the sequence of presentation is as follows: First, the DDL facility will be discussed. Specifically, memory considerations are discussed in order to explain why it was made a separate program. In addition, the functions allowed to be performed by the DBA are presented. Secondly, decisions made concerning the design and implementation of the editor are presented. This will entail a discussion of the functions allowed by the editor. In addition, the concept of "user

friendliness" is discussed with respect to the implementation of the editor. The design and implementation of the access structure are also presented.

INITIAL DEVELOPMENT

During the original development of the AFIT Relational Database System, memory considerations were extremely important. Since there was an extensive amount of code developed by Mark Roth, the code was separated into segments. Each segment basically contained code in which all the procedures performed a logical function. For example, the Define segment (Ref 9) consisted of code used to define domain and relation definitions.

At the end of Mark Roth's thesis effort, the system consisted of seven segments: the maximum allowed by UCSD Pascal. Consequently, at the beginning of this thesis effort, memory space was thought to be a critical resource. As a result, various means of obtaining this extra space was considered, resulting in the decision to free several existing segments so they could be used for development.

In order to free segments, it was decided to take what Mark Roth termed the DDL Processor (Figure 2) and make this a separate program. In addition, the functions referred to as the Boss functions, Inventory and Initialize, were to be included in this facility. The result was a program which could only be accessed by the DBA by means of a special user identification number. This

program allows a means to define and destroy domain and relation definitions. In addition, the DBA can get a full listing of the contents of the domain and relation definitions. This privilege is not allowed to any other user.

As a result of making this facility a separate program, two segments were freed for use. These were the Define and Inventory Segments (Ref 9). After their removal, the remaining system could be viewed as in Figure 9.

In this system, the user is no longer given the privilege of defining his own domain or relation definitions. Also, as previously stated, the user is not allowed a full listing of the domain or relation definitions attached. Specifically, the security identification numbers are not listed.

From Figure 9, it can be seen that several modules still needed to be implemented. These modules are the starred boxes. Their implementation will be discussed in the following sections.

IMPLEMENTATION TECHNIQUES AT THE DATA ENTRY LEVEL

During previous thesis efforts and during this thesis effort, interest was expressed concerning the implementation of an appropriate interface at the data entry level. The concern for an appropriate interface is well-founded, since the database system is specifically being implemented for teaching and research purposes.

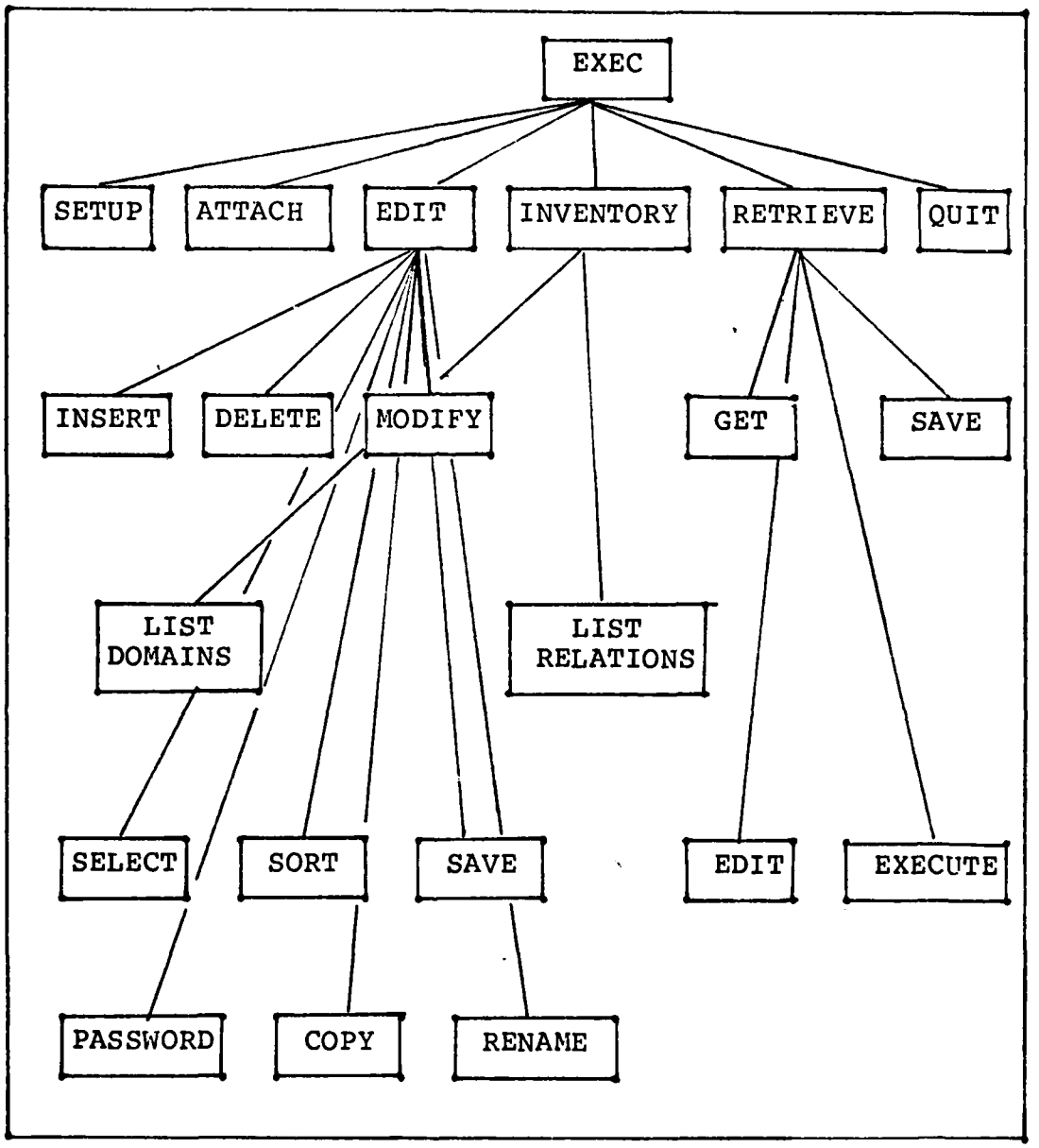


Figure 9. Roth's System without the DDL

Since this is the case, the implementation of a user interface that is not too time consuming to use would be an advantage. As a consequence, some of the concerns expressed in previous efforts were considered when selecting the relational algebra as the basis for the design of the Data Manipulation Language and the Data Definition Language. However, during this thesis effort special concern is expressed about the actual interaction between the machine and the user. Specifically, concern was expressed about the design of an appropriate interface which allows the user to convey to the editor the operations they would like performed.

Machine/User Interaction at the Edit Level

The desire for an interface that is "user friendly" stems from the following assumption which was found important in designing any language (Ref 17: 38):

> "A user cannot be expected to become familiar with a thick user manual in order to start using a language."

This statement has several ramifications. One, this statement suggests the need for an easy to learn formal high order language. However, this statement also subtly suggests the need for a user interface that aids the user in interacting with the system. Specifically, in the design of the Editor, this viewpoint was explored. As a

result, the Editor makes an attempt to interact with the user in a "friendly" manner. For instance, the user is prompted for all the information necessary to perform an edit operation. This is in lieu of the user being taxed with the task of inputting an edit command that must adhere to a rigid format. In addition, the user is no longer required to remember small details concerning the structure of a relation such as the attribute name. This is displayed as a prompt for the user.

An additional feature is added in order to enhance the "friendliness" of the editor and to maintain the integrity of the database. Specifically, all information entered by the user is checked before the operation desired is actually performed. In particular, the data entered is checked to see if it meets syntax specifications. Also, it is checked to see if data adheres to constraints specified when the relation was defined. If not, the user is given the option of correcting the information entered. Once all the information is entered correctly, the resulting command is displayed in its entirety. This is so the user can verify that the operation desired is actually the operation that will be performed by the system. If so, the operation is carried out and if properly completed, a message is displayed on the CRT indicating this fact.

Not only was the language interface given special consideration, but a format for display was also

considered. It was suggested that a predefined screen format would aid in making the system easier to use. As a result, a screen format was determined as shown in Figure 10.

Edit Commands Chosen.

As mentioned in Chapter 3, the process of deciding what edit commands to implement and how to implement these commands brought up an interesting question. Particularly, the question concerned extending the DML as designed by Mark Roth. It was decided that provisions should be made so that future implementations could include an extended DML.

Originally, that portion of the DML that performed the update function included the update operations as shown in Figure 9. However, it was decided that it was inappropriate to include most of these commands as update operations. For example, the Save, the Password, the Resort, and the Rename commands were thought to be inappropriate because their use by a user other than the DBA may compromise the integrity and security of the database system. For example, the Save command, as described by Mark Roth, would cause relations created as a result of relational operations on other relations to become permanent. However, it was previously decided that only the DBA would have control of the creation of permanent relations. Thus the Save command would be in violation of this rule. As for the Rename command, the

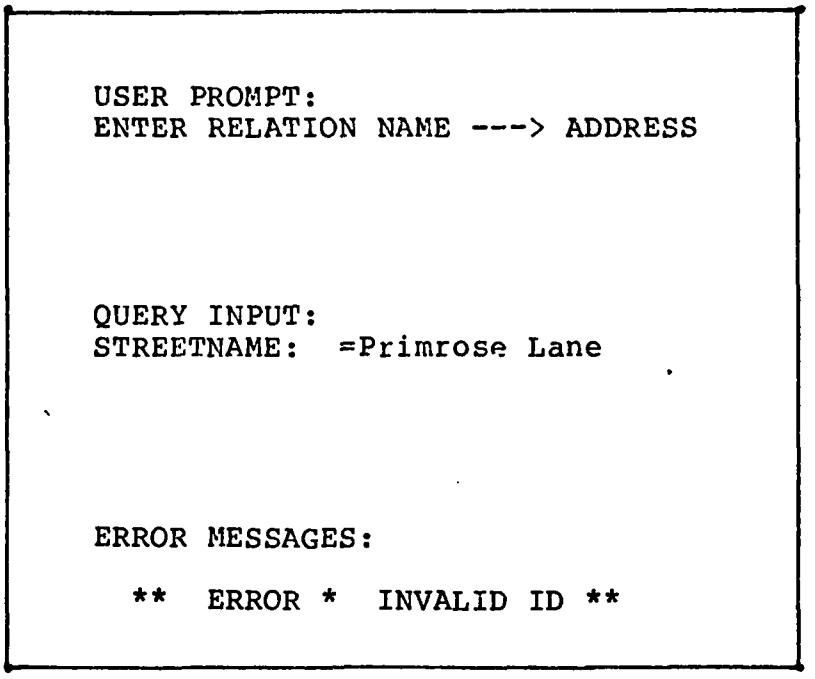


Figure 10. Screen Format

Resort command and the Password command, these too were felt to be functions which should only be allowed use by the data base administrator. As a result, they were eliminated. The Select command was also not needed.

The Copy command was kept because it offered the user a faster means of putting existing information into a relation without retyping the tuples. However, the function was changed. Now the function of the Copy is to insert a tuple from one relation into another predefined relation. An additional command was added, the Transfer command. The transfer command moves a tuple into a relation from a source relation and deletes it from this source relation. The insert, modify, and delete commands

were retained as described by Mark Roth (Ref 16: 119-120).

DESIGN OF THE LOW LEVEL ACCESS STRUCTURE

In order to provide some direction for the design of the low level access structure, it was necessary to do some additional research concerning the actual physical representation of information on secondary storage. Specifically, it was necessary to understand what decisions had to be made about the block and record structure. Also, it was necessary to understand some of the fundamentals about the basic file types available. As a result of this research, additional advantages of Theo Haerder's access structure were realized. The findings of this research are discussed in the following section:

Findings of Research

During the initial design phase of the access mechanism, it was determined that some means of organizing a relation file on disk must be formalized. From the research, it was decided that three types of file organization should be considered for the implementation of the low level access mechanism. These are the index sequential, the index file, and the sequential file. In researching these three file types, two characteristics specifically came to attention which aided in making the final decision as to what file organization to recommend. The first concerned the fact that the use of an index file is sufficient when there is no requirement to provide for

efficient sequential access. This allows a record to be placed any where in the relation file as long as a pointer exists in some index that allows the record to be fetched. Thus, file maintenance is considerably easier as compared to where data has to be placed in a certain order. An additional factor affecting the decision concerning file organization was that with an index file no overflow area is really needed. However, with an index file, the major maintenance problem is caused by the need to update all the indexes that refer to a record whenever a record has been added, deleted or moved.

It was determined that Theo Haerder's access structure is an index into a relation file where the file can either be organized sequentially or not. As pointed out, however, no restriction on the placement of a data record exists if the file is not sequentially organized. Consequently, it was thought that the file format should be non-sequential. To further justify this decision, the following factors were also considered: One, Theo Haeder's design allows for the access of any record by any attribute which is defined over some existing domain for which a B-tree exists. Thus, there is no reason to organize the file sequentially on some key value. This technique would just allow another means of accessing information. However, if the information had to be accessed on a different attribute other than the key, the B-tree index would still have to be accessed. In addition, Theo Haerder's design allows for

sequential access of a file if the leaf nodes are organized as he describes. Thus, fast sequential access is provided on any attribute instead of just the key attribute.

It should be pointed out that Roth allows for the user to define a directory for a certain attribute. This, as previously mentioned, is to decrease query processing time. However, this is not necessary with Theo Haerder's access mechanism because a means already exists for sequentially or randomly accessing information based on an attribute value defined on an existing domain. Specifically, random access is provided by searching the B-tree for a specific attribute value. Sequential access is provided by searching the leaf nodes of the appropriate B-tree.

When designing the low level access structure, it was also necessary to review the fundamentals concerning records and blocking. Briefly, records are the actual units of data storage on the logical or file level. The fitting of records into blocks is referred to as blocking (Ref 19 : 44). This investigation revealed that there are three methods of blocking. These are the following:

1. Fixed blocking for fixed length records
2. Variable length blocking, spanned
3. Variable length blocking, unspanned

Between these three methods, certain tradeoffs exist. Specifically, when deciding on the type of record to use, the tradeoff between ease of programming and wasted space was taken into consideration. Particularly,

it was thought that spanned variable length blocking would be the best to use in terms of efficient use of space. However, in terms of ease of programming, fixed records were thought better. Specifically, by using fixed records, it would be easier to identify the beginning and end of a record and thus simplify the design and implementation of the access method and file format. Otherwise, another means of identifying the beginning and end of a record must exist. For example, a length indicator can be positioned before every record so that the beginning point of the body of the next record can be reached by skipping over the body of the current record. Another alternative is to maintain a table for each block which gives all the record positions in a block. Both alternatives, however, increase the amount of information to be stored and also cause more processing to be done in order to a update a file (Ref 19: 46).

THE ACTUAL DESIGN

The actual design of the access structure included a design of the interface between the editor and the access structure. The technique used was top down in nature in order to determine the processing flow. After the design was completed, it was decided that the actual implementation of portions of the editor could be revised to provide for a more flexible means of expressing update requests to the system. The design of the access mechanism is discussed in the next sections. The sequence of

presentation is as follows. The first section presents the basis for the data structures used. The second section presents the design of the insertion modules. The third section discusses the design of the deletion modules and the forth, the design of the modification modules. In each section the interface between the editor and these modules are presented.

The Data Structures Used

In order to implement the deletion and insertion algorithms, it was necessary to determine the types of files necessary and to define some record structure for each file type. It was decided that there would be three basic file types used in order to store relations and implement the B-tree data structure. Specifically, a file would exist for each relation. This file is referred to as a relation file. To store the B-tree, two types of files were decided upon. One was to contain the internal nodes of the B-tree. The second file contains the leaf nodes of the B-tree. In order to design the record format for each file, it was decided that fixed records would be better to use because it would facilitate programming. Thus, in all three files a record is of a fixed length. The record structure of each type of file is discussed in the following sections.

The Btree Record Structure

Decisions concerning the B-tree record format were based on several factors. First, it was decided that a

block (512 bytes) would constitute a B-tree node. This number is based on the standard UCSD Pascal block size. As a consequence of this fact and the additional constraint that fixed records were to be used, the size of a B-tree record had to be, at the maximum, 512 bytes. However, this would never be the case since the maximum domain size would be 80 characters and thus the maximum record size would be 128 bytes. The means for determining this boundary is given as follows. The record size of a particular B-tree is fixed. However, the record size of different B-trees may vary depending on the domain size, which is a string variable that could have the maximum length of 80 characters. To determine the maximum record length, the maximum domain size is added to the length of the remaining fields. In this case, this value is 24 (see Figure 11). Adding these two values, the maximum record size is equal to 104. With this information, it was decided that the record size should be at least 104 bytes, but should divide evenly into 512 and thus leave room for extra fields if deemed necessary. The smallest value greater than 104 that evenly divided 512 was 128. Consequently, the minimum fanout ratio is four. However, the fanout ratio could be greater than four, depending on the domain size.

The B-tree record format is given in Figure 11.

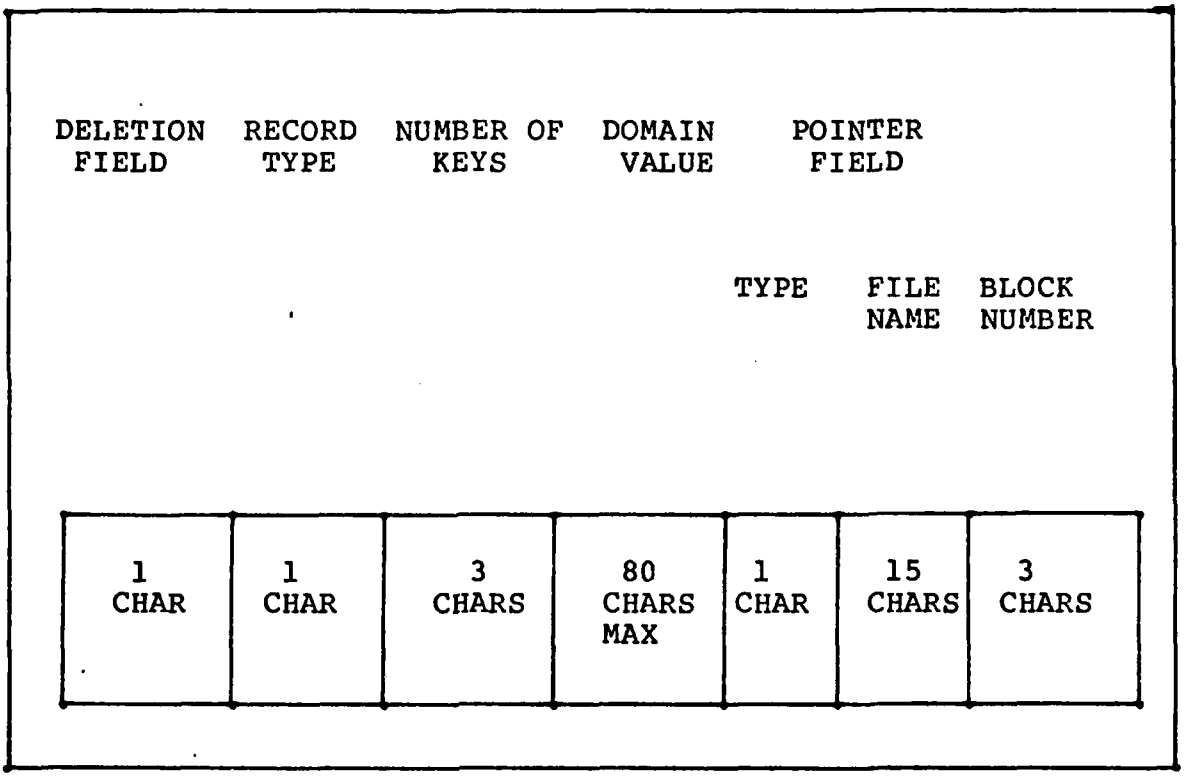


Figure 11. B-tree Record Format

An description of each field is given as follows:

Deletion Field: a field which indicates whether or not the record has been deleted

Record Type: a field which indicates the type of record. In this case B/H.

Number of keys: the number of keys in the B-tree node.
Domain Value: the domain value for the record

Pointer field: a pointer to either a B-tree node or a leaf node. It consists of type field (B/L), a filename, and a block number.

There is an additional B-tree record type. This record is contained in the B-tree file header. The fields are shown in Figure 12.

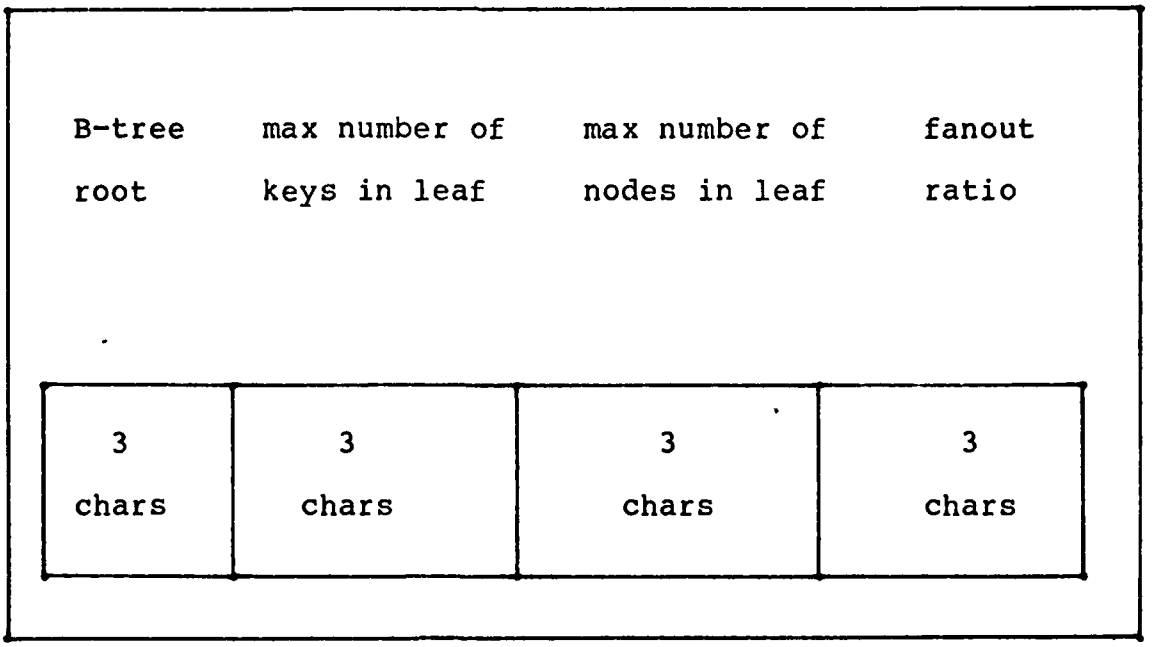


Figure 12. The B-tree header record format

An explanation of each field is given as follows:

B-tree root: a field which contains the block number of the the node which represents the B-tree root

Maximum number of keys in leaf : a field which contains the maximum number of the keys in a leaf node

maximum number of nodes in leaf : a field which contains the number of pages in a leaf node

fanout ratio: a field which contains the fanout ratio or maximum number of records in a B-tree node

The Relation File Record Structure

Two types of records are suggested for the relation file. The first record is called the relation file header record and has the following format which is shown in Figure 13:

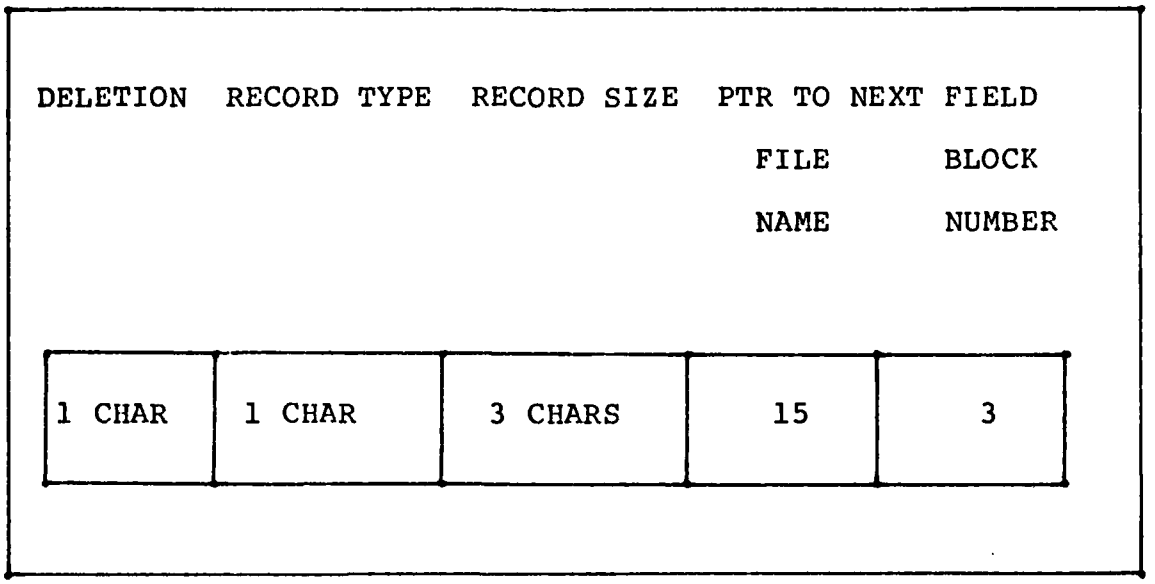


Figure 13. A Relation File Header Record

A description of each field in the record is given as follows:

Deletion field: a field indicating whether or not the record has been deleted

Record type: a field indicating the type of relation file record (T/H)

Record size: a field containing the maximum number of characters in the record

Pointer to: next file a field indicating a pointer to the overflow file

This record should be placed in the first block of the relation file header. The field which contains a pointer to the next file could be used to link the file to to an overflow file. The second record format is shown in Figure 14.

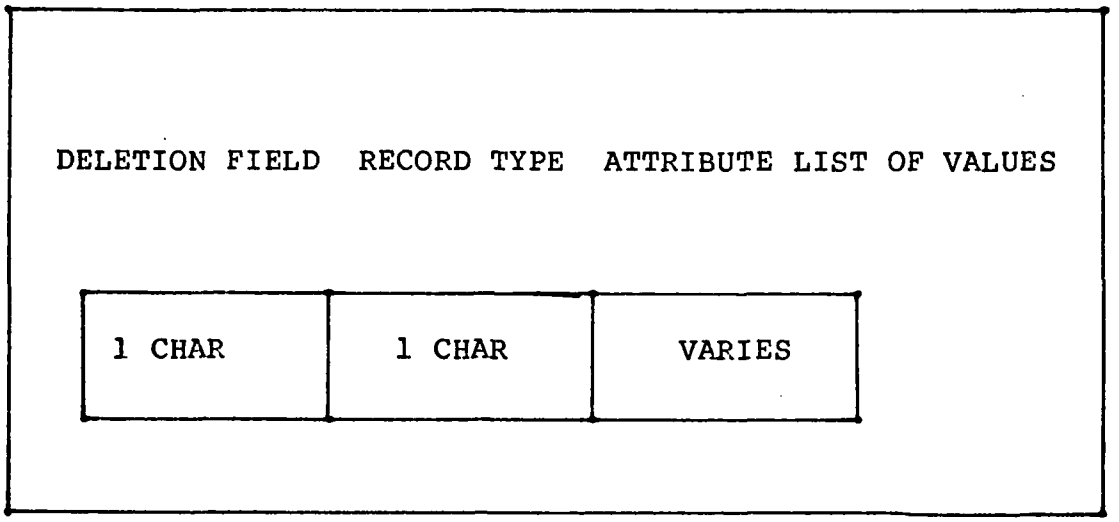


Figure 14. The File Record Format

An explanation of the fields in the file record are given as follows:

Deletion Field: a field indicating whether or not the record has been deleted

Record Type: a field indicating the type of file record

Attribute list: list of attributes values in the tuple

The Leaf Record Format

Three types of records are suggested for the leaf node file. The first record contains pointer information. This pointer information includes an overflow pointer which consists of a file name and a block number in which the overflows of a leaf node are placed. Other pointer information includes a pointer to the next leaf node and a pointer to the previous leaf node. Other information contained in this record is indicated in Figure 15.

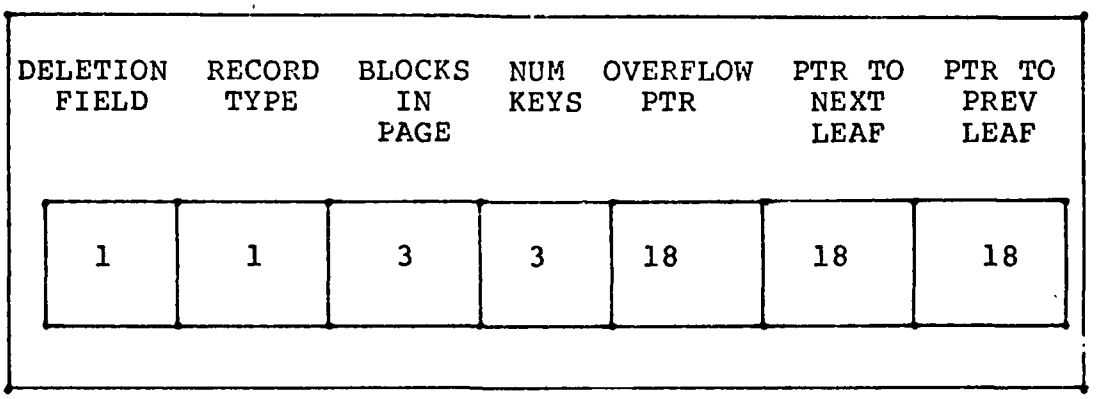


Figure 15. The leaf node pointer record

The second type of record suggested is a key record. This record is used so that the domain value does not have to be kept in every TID record. The record format is shown in Figure 16.

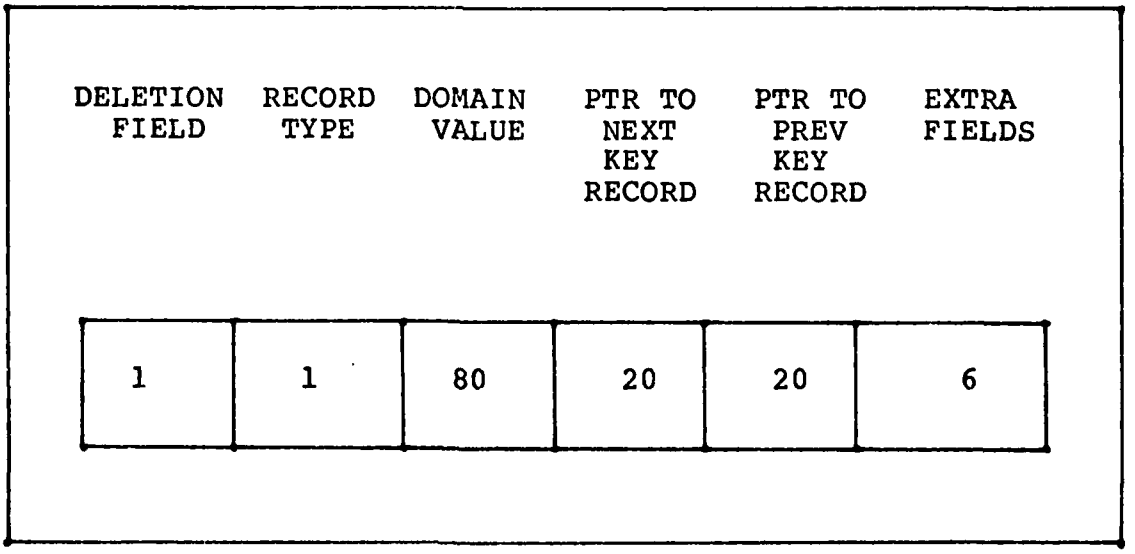


Figure 16. The Leaf node key record

A description of each field in the record is given as follows:

Deletion field: a field indicating whether or not the record has been deleted

Record type: a field indicating the type of record

Domain value: a field which contains the domain value of the key record

Next key pter: a field which points to the next key record

Prev key pter: a pointer to the previous key record

The final record type is a TID record. This record contains the logical address of a tuple with a particular attribute value defined over a domain. The TID record has the following format shown in Figure 17.

DELETION FIELD	RECORD TYPE	RELATION NAME	FILE NAME	BLOCK NUM	RECORD NUM	ATTRIBUTE NAME
1 CHAR	1 CHAR	15 CHARS	15 CHARS	3 CHARS	3 CHARS	80 CHARS

Figure 17. The leaf node TID record

A description of each field is given as follows:

Deletion field: a field which indicates whether or not the record has been deleted

Relation field: a field which contains the relation name of the attribute

TID: a field which contains the logical address of the tuple which contains an attribute of a particular value

Attribute name: the attribute name of the key value

Additional Data Structures

Additional data structures were designed and implemented so as to serve as pointers to various nodes in the Btree index. Specifically, the type, Treeptr, was declared to be a record which consisted of a file name, block number, and pointer type. The pointer type could either be a "B" , for a B-tree node, or "L", for a leaf node. This structure is used in the same manner as Pascal pointers to traverse trees and to locate nodes. Another structure defined was ATTRECORD. This was a record used to contain information about the new attribute value inserted or deleted from a B-tree. Specifically, it contains the following information:

ATTRIBNAME: the attribute name
ATTRVALUE : the attribute value
DOMAINTYPE: the domain type of the attribute
DOMSIZE: the maximum size of the domain value
RELNAME: the relation to which the attribute belongs

Design of the Insertion Modules

Basic Function of the Insertion Operation

The insertion of tuples into a relation file is a basic edit operation. The function of the insertion process is to insert a tuple into a relation file and update the access trees which correspond to each attribute of the tuple to be inserted. The editor provides a means of entering the tuple from the terminal and stores this information in a structure in primary memory so that it can be passed to the insertion modules. In order to perform the insertion operation, the editor calls a module called insert. The general algorithm for the insert modules is as follows:

INSERTION ALGORITHM

```
Begin
  If not a Duplicate Tuple then
    begin
     Put the Tuple in the Relation File
     For each Attribute in the Tuple
      Update the corresponding Btree
    endif
End
```

The basic structure of this module is given in Figure 18.

A description of those modules shown is continued in the following sections. The starred boxes are modules which have not yet been implemented.

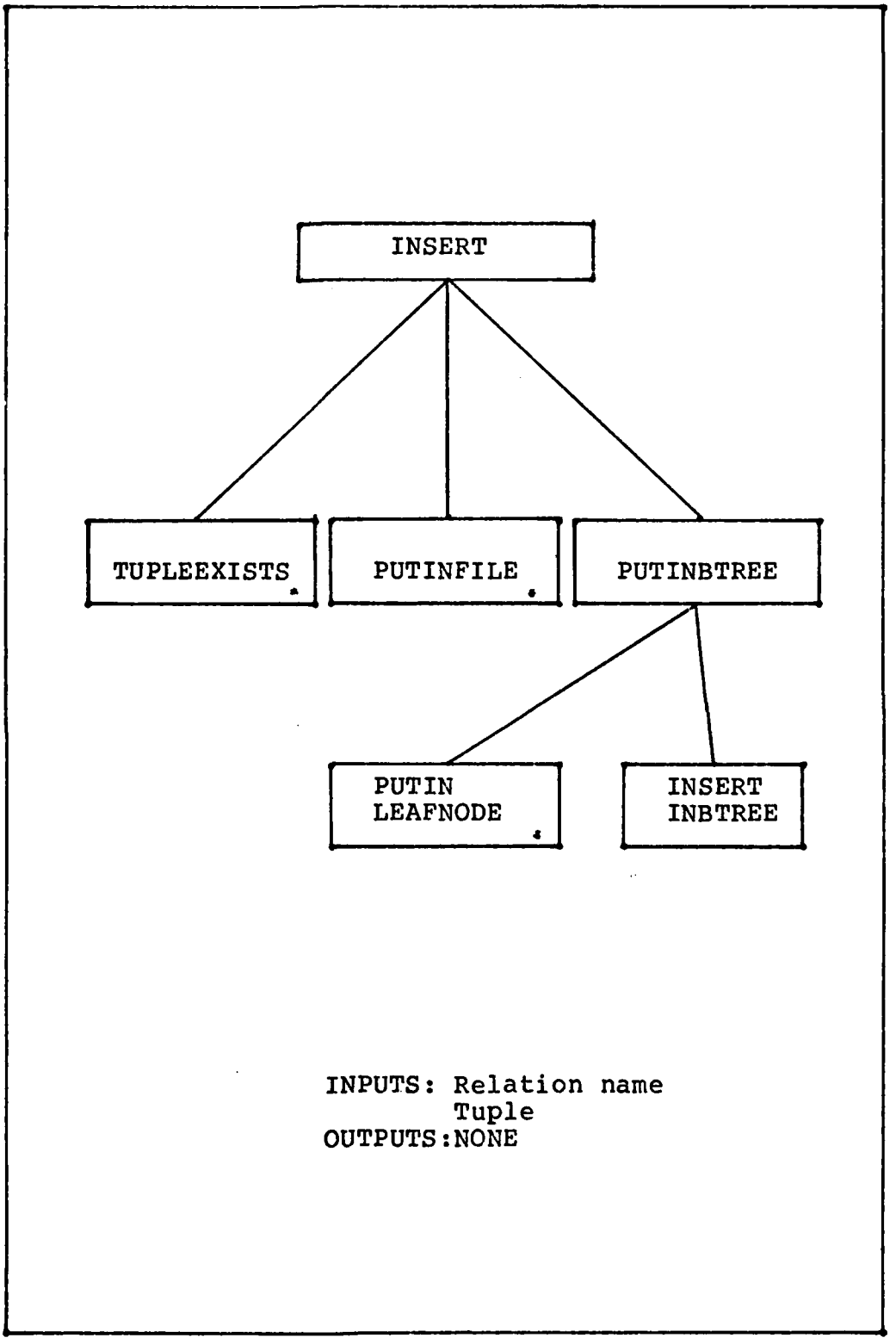


Figure 18. The Insert Module

The Tupleexists Module.

The function of this module is to determine whether or not the tuple to be inserted is a duplicate tuple. The general algorithm for this module is as follows:

```
TUPLEEXISTS ALGORITHM
Begin
  If the Btree exists then
   Begin
      Find the leaf node for the
      primary key attribute
      Search the leaf node
      If the value is found then
       tupleexists <-- true
      else
       tupleexists <-- false
   end
  else tupleexists <-- false
End
```

Special consideration has to be taken if the key is a concatentation of more than one attribute. Specifically, the B-tree has to be searched for the TID which corresponds to each attribute which forms the key. The TIDs retrieved from each B-tree searched must be the same.

The modular breakdown is shown in Figure 19. Note that to test if a tree exists, one needs to search the relation definition lists and find the attribute lists for the relation. Once this is found, find the domain pointer for this attribute and examine the file size for the B-tree. If the file size is zero, the B-tree does not exist on disk. However, if the file does exist, the B-tree root can be found in a record in the first block in the B-tree file. This contains information concerning the block number of the B-tree root.

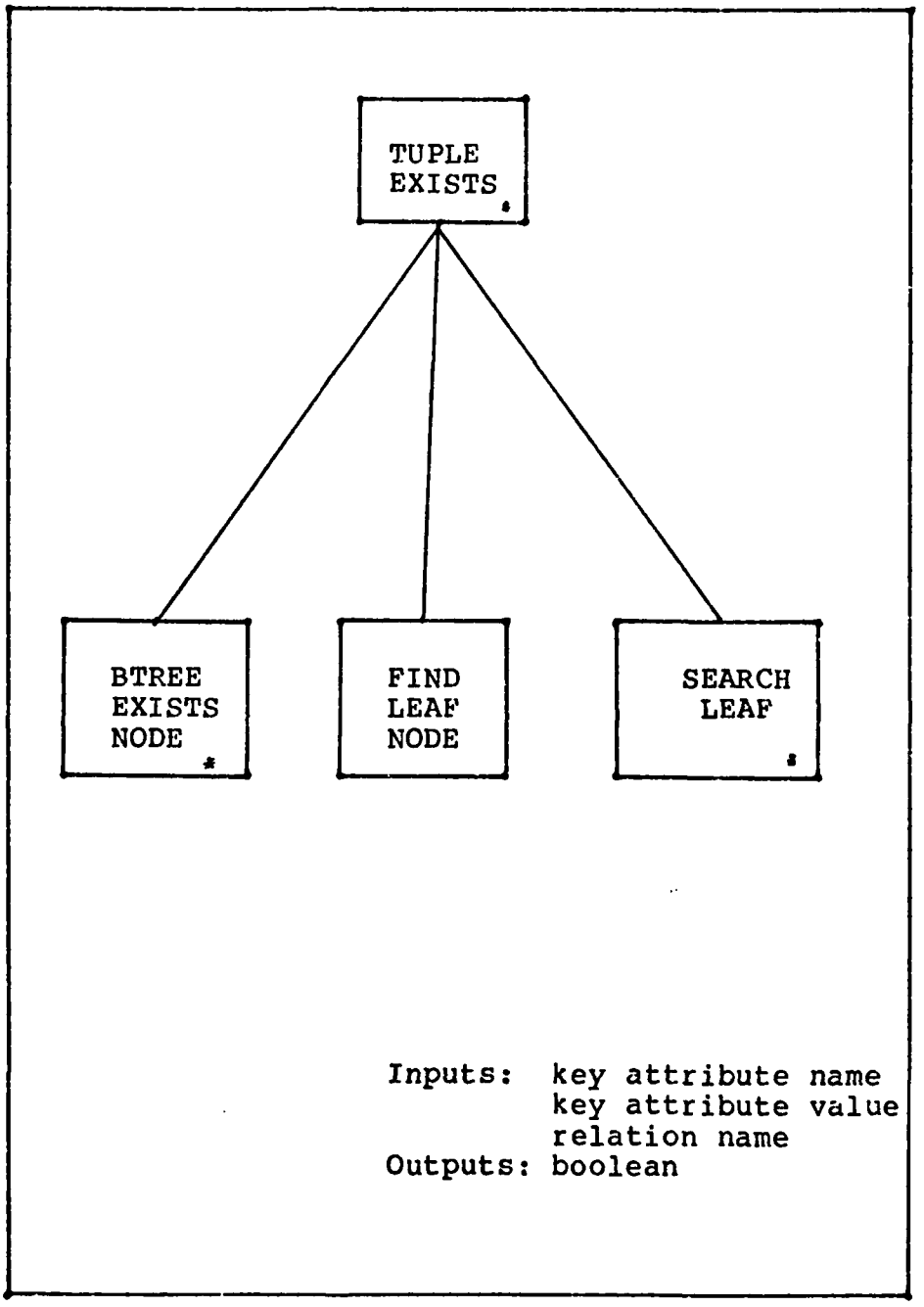


Figure 19. The Tupleexists Module

The function of the findleafnode module is to search the B-tree in order to find a pointer to a specific leaf node. This module is discussed in detail in the section, The Findleafnode Module. The searchleafnode module searches a leaf node to determine if the tuple exists. The tuple exists if the key value can be found in the leaf node.

The Findleafnode Module

The function of this module or procedure is to traverse the B-tree in order to find a pointer to a specific leaf node.

The basic algorithm for this module is recursive in nature. It is given as follows:

```
FINDLEAFNODE ALGORITHM
Begin
  If pointer is not pointing to leaf node then
     Read Btree node from disk
     Get the next node pointer from Btree node
     Findleafnode
  Else
     If not a new leaf then
       Set pointer leaf to present node pointer
     Else
       Assign new leaf file name
       Set block number in leaf pointer
     Endif
  Endif
End
```

A structural diagram of this module is given in Figure 20.

An example of this code's more subtle uses is given as follows. The value newleaf is set in getnextnodeptr. Suppose, the B-tree in Figure 21 has been constructed.

To insert the value 75 into the tree, a newleaf would

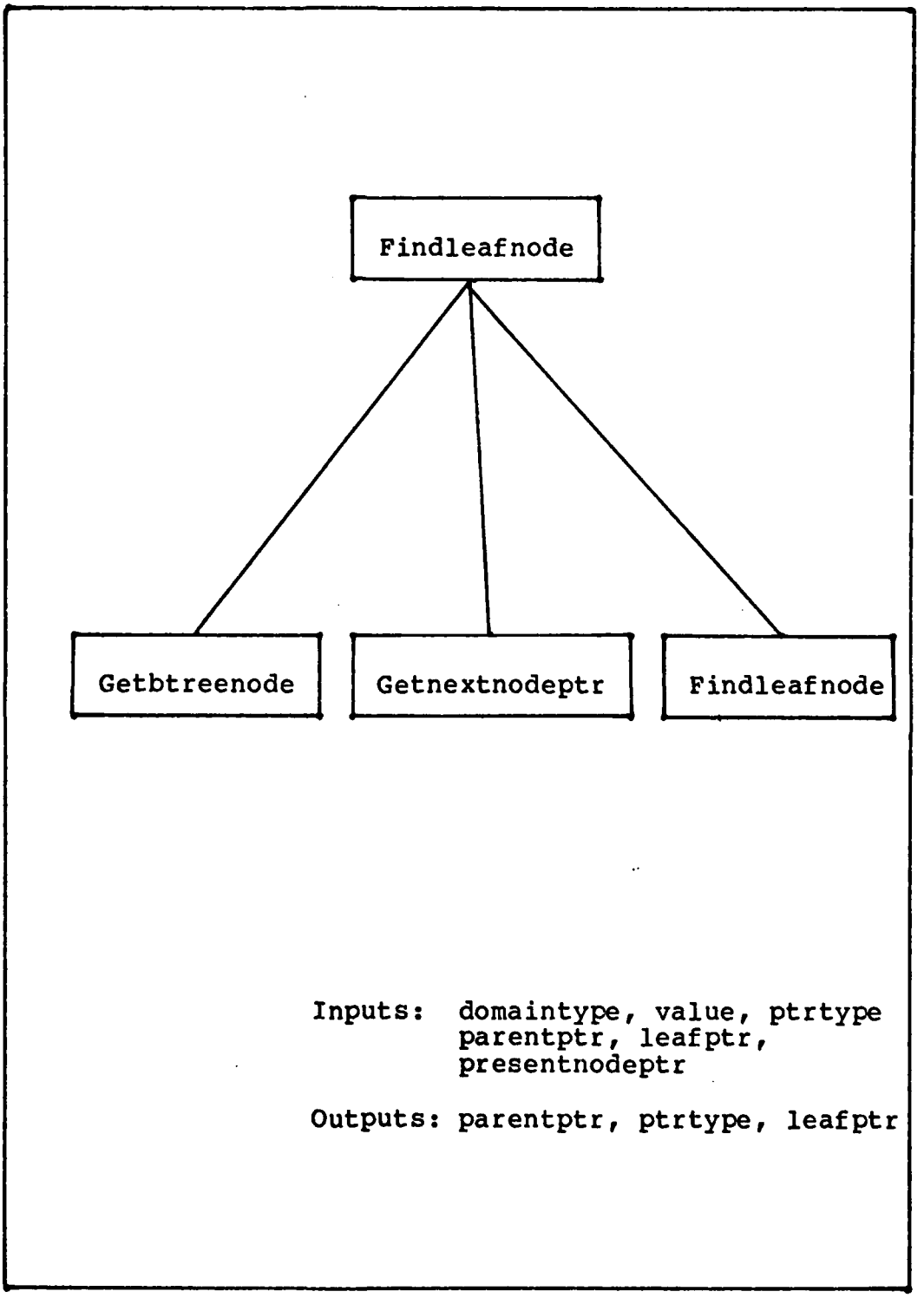


Figure 20. The Findleafnode Module

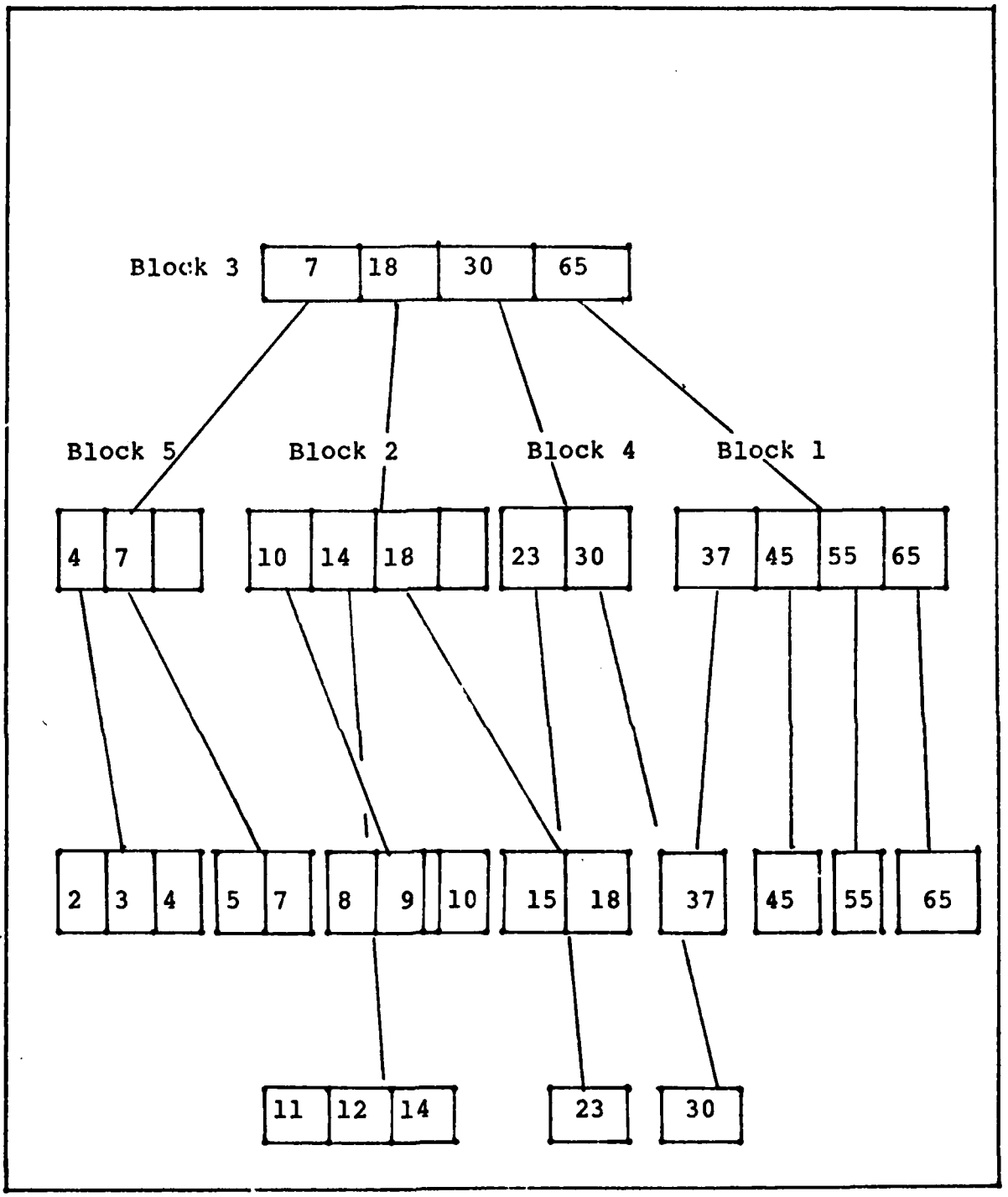


Figure 21. An example of the findleafnode algorithm

have to be created with the value 75 in it. However, a means of determining if a new leaf node needs to be created has to be established. This is accomplished while traversing the tree to determine into which leaf node to insert the value. Specifically, the variable newleaf is set if the value to be inserted is greater than any other value already in the tree and is not to be inserted in an existing leaf node. If newleaf is true, a new leaf node is to be created. Otherwise, the insertion is to take place in a existing leaf node.

The Putinfile Module

The function of this module is to determine the position of a free space in a relation file to insert a tuple and to place the tuple into this free space. This free space may be located at the end of the file or it may be a position in which a tuple has been deleted. This module should return a pointer to this tuple in the relation file so that it can be used when updating the B-tree. This pointer should consist of a filename, a block number, and a record number. A general algorithm for this module is given as follows:

```
PUTINFILE ALGORITHM
Begin
 Get the relation file name
 If the file is new then
   initialize the file
 Read blocks until an empty record is found
 Put tuple into a record
 Insert record into a block
 Rewrite block to disk
end.
```

A structural diagram for this module is shown in Figure 22.

The Putinleafnode Module

The function of the putinleafnode module is to insert new attribute values with their corresponding attribute and relation names into a leaf node. The attribute and relation names are also inserted to distinguish the value inserted from other attributes of relations with the same value. This will entail finding the correct leaf node in which the new attribute belongs. Once the correct leaf node is found, it must be determined if there is room to insert the new key value. This is accomplished by having a leaf record which contains a field which indicates the number of keys in the node (refer to Figure 15). The recommended maximum number of keys allowed in the leaf node is either equal to or greater than the fanout ratio of the B-tree itself. The reason is that if the number of keys allowed in the leaf node is less than the fanout ratio of the B-tree, then the number of domain values in the B-tree will be reached and the tree will grow at a much faster rate and thus may increase processing time. This may be illustrated by considering the B-tree in Figure 23 in which the key, 8, is inserted into the leaf node which is only allowed the maximum number of three keys.

This causes the leaf node to split and a new key value to be placed in the B-tree node (Figure 23B). However, if only four key values were allowed, the leaf

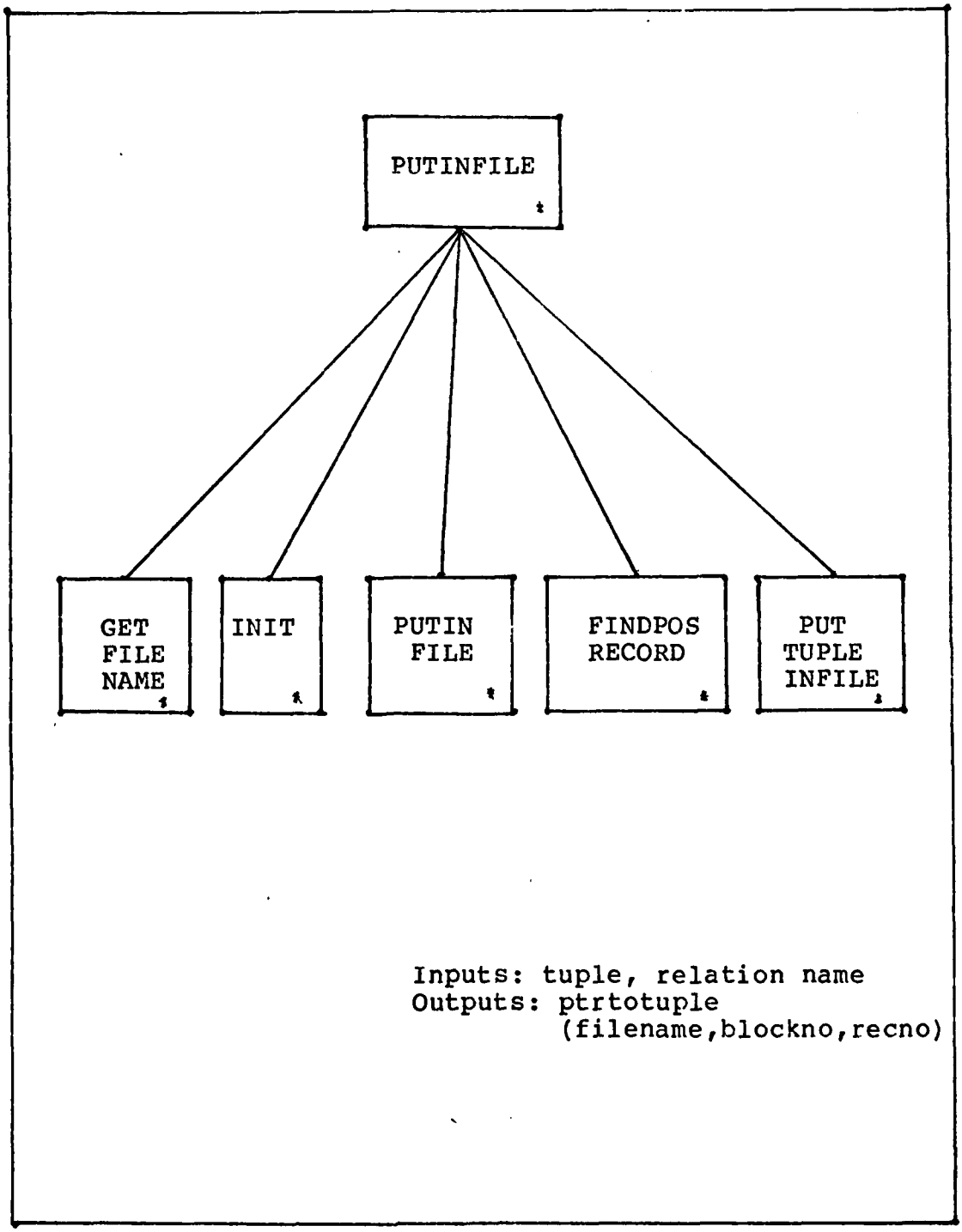


Figure 22. The Putinfile Module

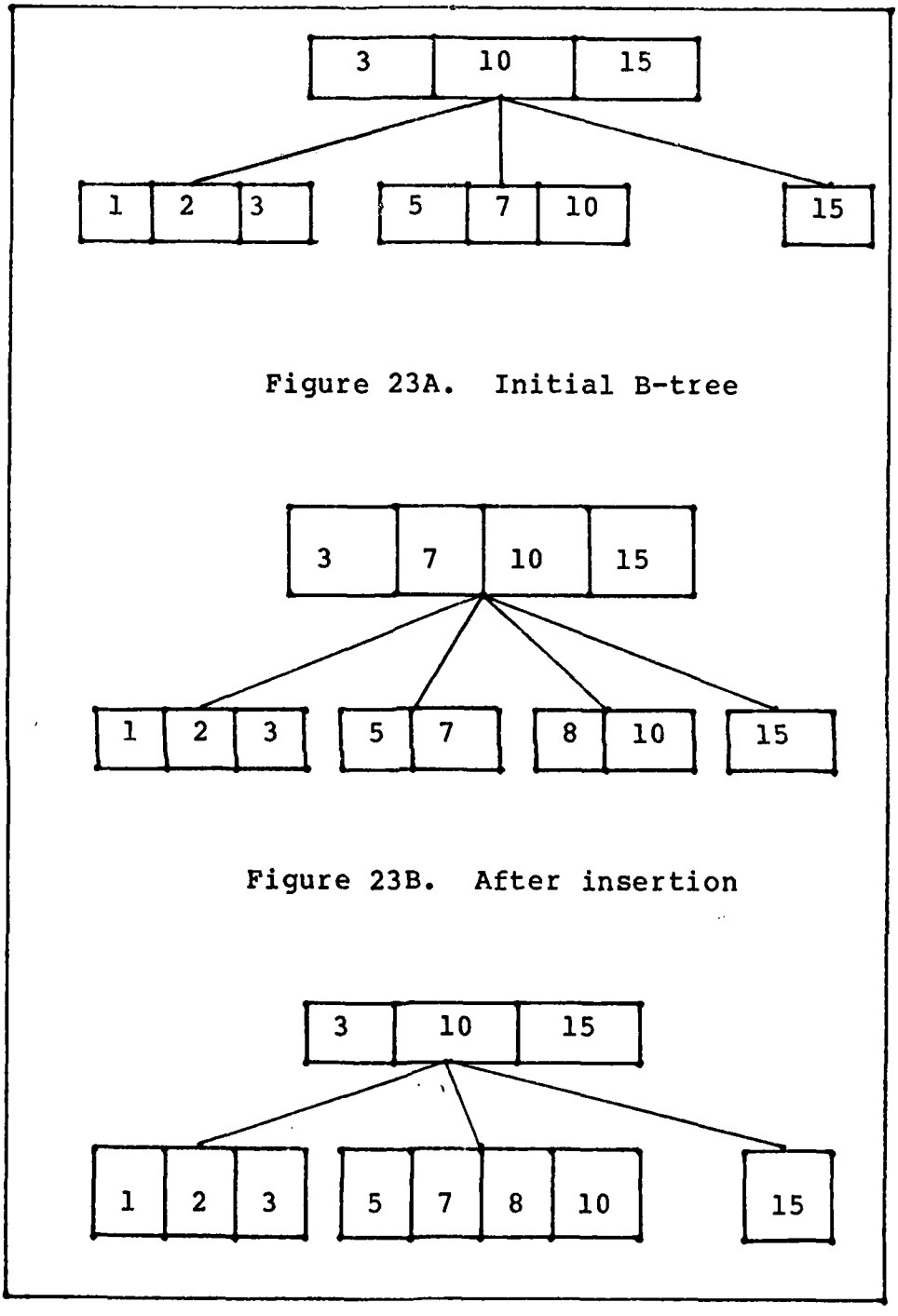


Figure 23A. Initial B-tree

Figure 23B. After insertion

Figure 23C. After insertion

node would not have to be split at this time, and the B-tree would not grow at this faster rate (refer to Figure 23C).

If there is room for the insertion, the position at which to perform the insertion must be determined. Basically there are two cases to be aware of for inserting a key value into the leaf. One, is when the key value to be inserted is a new key value, meaning that there is no attribute defined over that domain in the data base system. Case two is if the key value to be inserted is a value that already exists in the leaf node but is a different attribute of some defined relation.

In the first case, the insertion is rather simple because in this case no overflows should result. However, in the second case an overflow of the leaf may occur and some means of handling overflows should be devised. Haerder's diagrams suggest reorganizing the leaf node so that the last key values that do not fit into the leaf node are put in an overflow leaf node. A general idea of how the leaf nodes may be chained together is given in the Figure 24. The record structures of the leaf node will be discussed in a later section.

The Insertinbtree Module

The call to insertinbtree is only made if the new attribute value inserted in a leaf node causes a split in the leaf node, because it is the first attribute with this key value and there is no room in the leaf for insertions, or if a new leaf node is to be created because the value

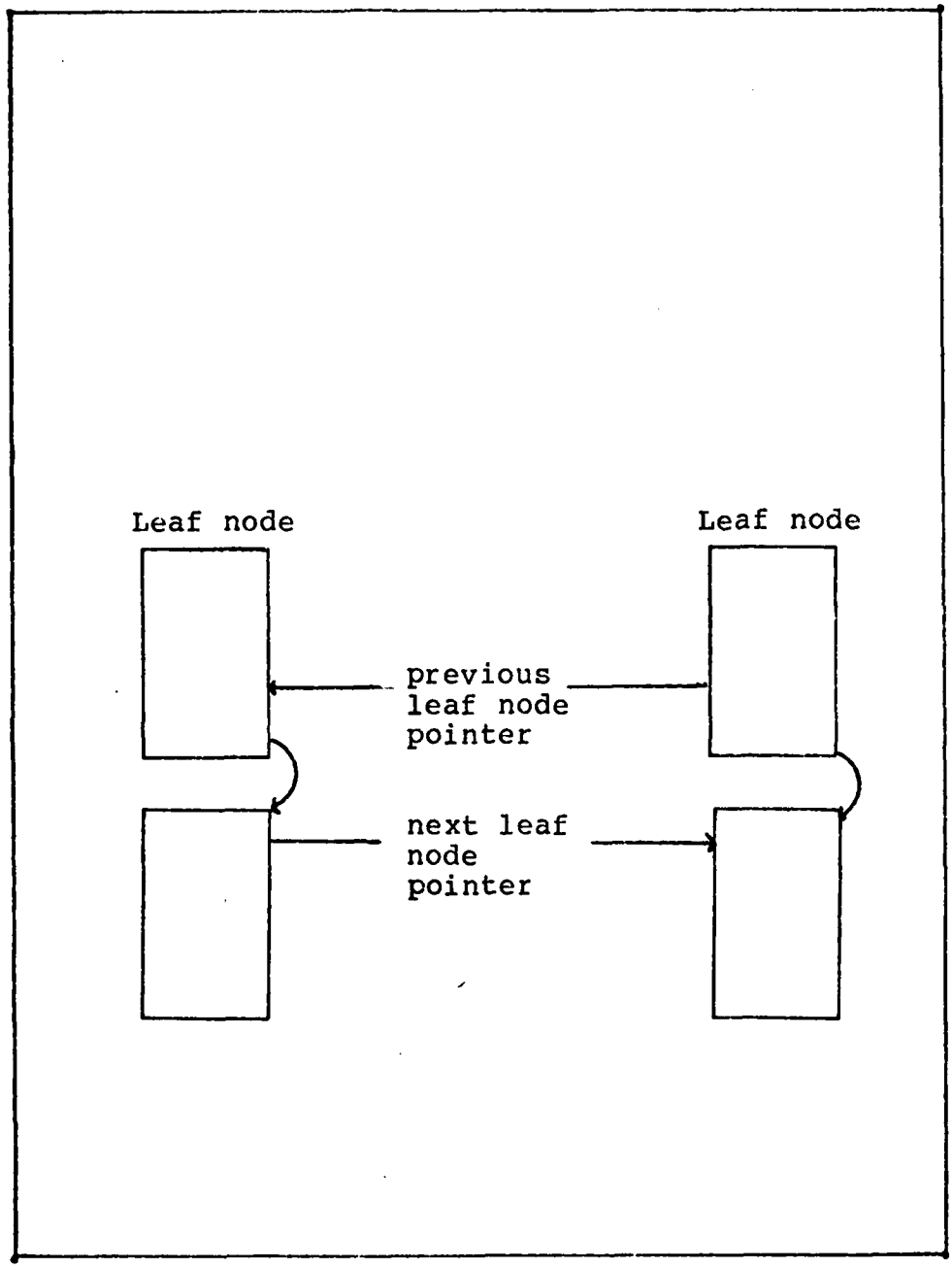


Figure 24. Leaf node chaining

is greater than any other value in the tree. The function of the insertinbtree module is to update the Btree so that it remains balanced and can serve as an index to the leaf nodes which in turn point to the actual tuples in a relation by way of a logical address which consists of the relation file name, the block number in the relation file, and the record number (see Figure 17). The inputs into the insertinbtree module are explained as follows.

INSERTINBTREE

Inputs:	Btreeroot:	a pointer to the root of the B-tree
	Newleafptr:	a pointer to a newly created leaf node
	Newmaxvalue:	the maximum value of the newly created leaf node
	Oldleafptr:	a pointer to a leaf node that was recently split
	Oldmaxvalue:	the maximum value of the leaf node recently split
	Parentptr:	the pointer to the parent of the leaf node in which the insertion was performed
	Opstatus:	the status of the leaf node, valid values are 'split' and 'emptyandchangemax'

The actual parameters of the insertinbtree module are shown as follows:

INSERTINBTREE

Parameters:	Oldchildptr:	the pointer to a child node which is either a leaf node or a B-tree node
	Newchildptr:	the pointer to a child node which is either a

leaf node or a Btree node. This pointer points to a node that was newly created.

Oldmaxvalue: the maximum value of the old child node pointed to.

Newmaxvalue: the maximum value of the new child node pointed to.

Statusofchild: a variable that indicates the operation that was performed on the child node

Presentptr: a pointer to the present node being manipulated

Stopptr: a pointer to the parent node of the leaf node

Newattribute: information about the newattribute to be inserted

The General Algorithm for the Insertion Module

The general algorithm for this procedure is given by the following:

```
INSERTINBTREE ALGORITHM

  IF PRESENTPTR <> PARENT OF LEAF NODE THEN BEGIN
     GET THE PRESENT BTREE NODE FROM DISK
     SEARCH NODE FOR THE NEXT NODE POINTER
     INSERTINBTREE
  ENDIF

  GET THE PRESENT NODE
  GET THE STATUS OF THE PRESENT NODE
  CASE STATUS OF THE PRESENT NODE
    EMPTY: IF FREE SPACE AND MAX VALUE CHANGED THEN
             SET STATUS INDICATOR <-- TO ALRIGHT

           IF CHILD WAS SPLIT AND THE MAX VALUE CHANGED THEN
             CREATE A NEW BTREE NODE
             INSERT MAX VALUES OF THE CHILD NODE
           ENDIF

           IF THE CHILD WAS EMPTY THEN
             INITIALIZE A NEW BTREE NODE
             CREATE A NEW BTREE NODE
```

```
        INSERT MAXIMUM VALUE OF THE CHILD NODE
        WRITE NEW NODE TO DISK
      ENDIF

      IF THE CHILD WAS SPLIT THEN
        CREATE A NEW BTREE NODE
        INSERT MAX VALUES OF THE CHILD NODE
        WRITE NEW NODE TO DISK
      ENDIF

FULL: IF FREE SPACE AND MAX VALUE CHANGED THEN
        DELETE THE MAX VALUE OF THE PRESENT NODE
        INSERT MAX VALUE OF THE CHILD NODE
        WRITE PRESENT NODE TO DISK
        GET MAX VALUE OF THE PRESENT NODE
        UPDATE STATUS OF CHILD INDICATOR
      ENDIF

      IF THE CHILD WAS EMPTY THEN
        CREATE A NEW BTREE NODE
        SPLIT THE PRESENT NODE DISTRIBUTING VALUES
        GET MAX VALUE OF THE NODE CREATED
        GET MAX VALUE OF THE NODE SPLIT
        WRITE NEW NODE TO DISK
        WRITE NODE SPLIT TO DISK
        UPDATE STATUS OF CHILD INDICATOR
      ENDIF

      IF THE CHILD WAS SPLIT THEN
        CREATE A NEW BTREE NODE
        SPLIT THE PRESENT NODE DISTRIBUTING VALUES
        GET MAX VALUE OF THE NODE CREATED
        GET MAX VALUE OF THE NODE SPLIT
        WRITE NEW NODE TO DISK
        WRITE NODE SPLIT TO DISK
        UPDATE STATUS OF CHILD INDICATOR
      ENDIF

      IF THE CHILD WAS SPLIT AND MAX VALUE CHANGED THEN
        DELETE MAX VALUE OF PRESENT NODE
        CREATE A NEW BTREE NODE
        SPLIT THE PRESENT NODE DISTRIBUTING VALUES
        GET MAX VALUE OF THE NODE CREATED
        GET MAX VALUE OF THE NODE SPLIT
        WRITE NEW NODE TO DISK
        WRITE NODE SPLIT TO DISK
        UPDATE STATUS OF CHILD
      ENDIF

      IF THE CHILD WAS FULL AND THE MAX VALUE CHANGED THEN
        DELETE THE PRESENT MAX VALUE
        INSERT NEW MAX VALUE IN THE PRESENT NODE
        GET MAXVALUE OF PRESENT NODE
        WRITE THE NODE TO DISK
```

```
                    UPDATE THE STATUS OF CHILD
                  ENDIF

FREE SPACE:       IF CHILD HAD A FREE SPACE AND MAX VALUE CHANGED THE
                    DELETE THE PRESENT MAX VALUE
                    INSERT NEW MAX VALUE IN THE PRESENT NODE
                    GET MAX VALUE OF THE PRESENT NODE
                    WRITE THE PRESENT NODE TO DISK
                    UPDATE STATUS OF CHILD
                  ENDIF

                  IF CHILD WAS SPLIT THEN
                    INSERT THE NEW MAX VALUES
                    WRITE THE PRESENT NODE TO DISK
                    GET MAX VALUE OF THE PRESENT NODE
                    UPDATE THE STATUS OF THE CHILD
                  ENDIF

                  IF THE CHILD WAS SPLIT AND THE MAX VALUE CHANGED TH
                     DELETE THE PRESENT MAX VALUE
                     INSERT NEW MAX VALUES
                     WRITE THE PRESENT NODE TO DISK
                     UPDATE STATUS OF CHILD
                  ENDIF

                  IF THE CHILD WAS EMPTY THEN
                    INSERT MAX VALUE
                    WRITE THE PRESENT NODE TO DISK
                    UPDATE STATUS OF CHILD
                  ENDIF

                  IF THE CHILD WAS FULL AND MAX VALUE CHANGED THEN
                    DELETE THE PRESENT MAX VALUE
                    INSERT MAX VALUE
                    WRITE THE PRESENT NODE TO DISK
                    GET PRESENT NODE'S MAX VALUE
                    UPDATE STATUS OF CHILD
                  ENDIF
          ENDCASE

          IF THE PRESENT NODE IS THE ROOT NODE
             IF THE CHILD NODE WAS SPLIT
               CHANGE THE ROOT FIELD IN THE FILE HEADER


     END INSERTINBTREE
```

An Example of the Insertion into the B-tree

An example of how the insertion into the B-tree works is given by the following. Suppose the B-tree shown in

Figure 25 has been constructed. If the value, 5, is to be inserted into the B-tree, the first step is to insert this value in the appropriate leaf node. To find the leaf node, the B-tree is traversed and a pointer to the leaf node and its parent node is returned. The status of the leaf node is then determined by counting the number of key values in the leaf node. If the number of key values is less than the fanout ratio, then the leaf node has a free space in it and an insertion can take place. However, in the example the number of keys is equal to the fanout ratio, and thus it is full. However, this alone does not determine whether or not a value can or cannot be inserted into the node. If the value is a new key value to be inserted, it will cause a split in the leaf node. However, the leaf node has to be searched to determine if the new value to be inserted is a value of an attribute of some defined relation that has already been inserted in the B-tree. This is essentially done by finding the key value and searching all entries with that value to see if the attribute name and relation name match that of the value that is to be inserted. In the example, the value five is a new key value and therefore cannot be inserted into the existing leaf node. To insert the value, the leaf node must be split and the values are distributed between the newly created leaf node and the node split. This is shown in Figure 26.

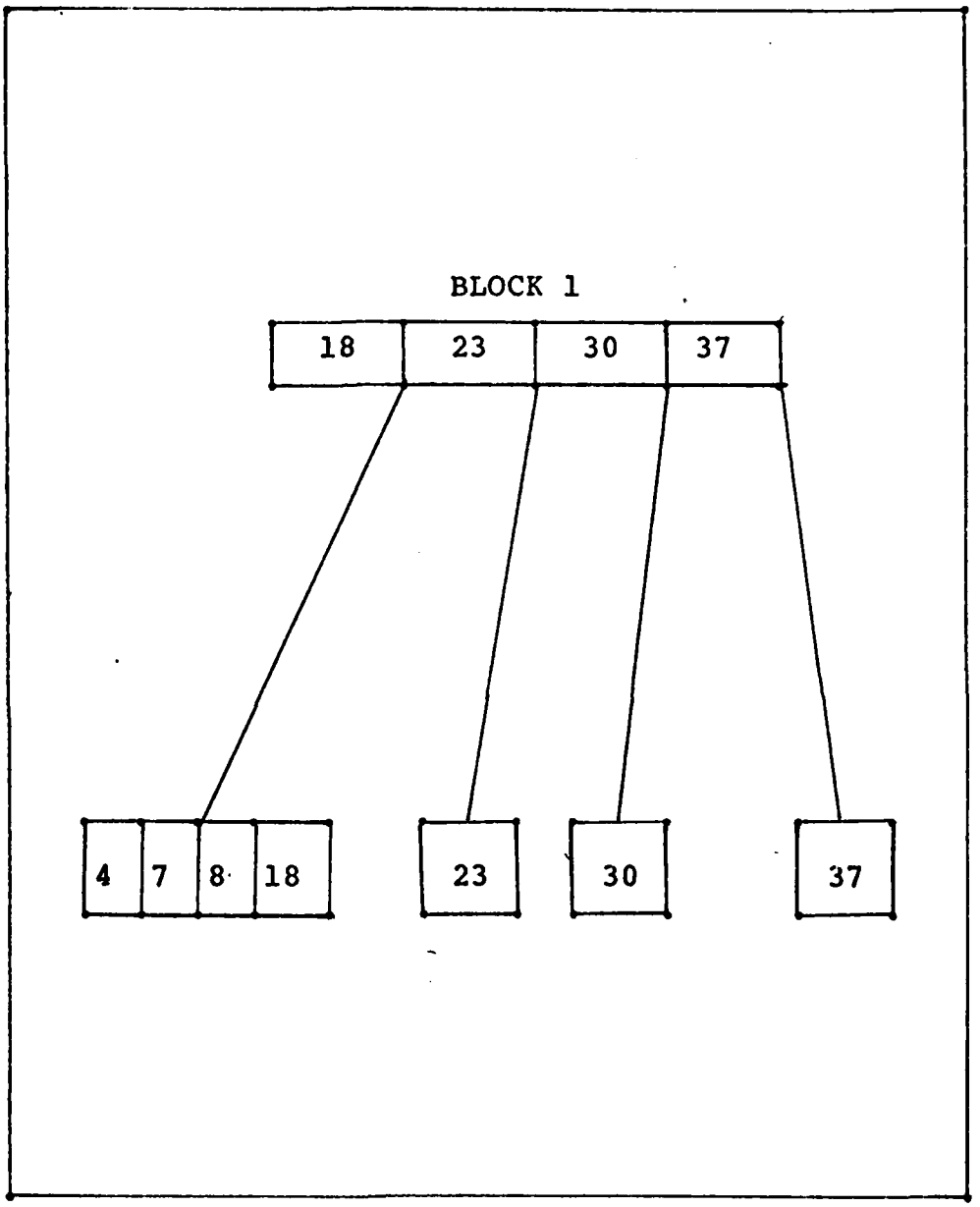


Figure 25. Example of an insertion into a B-tree

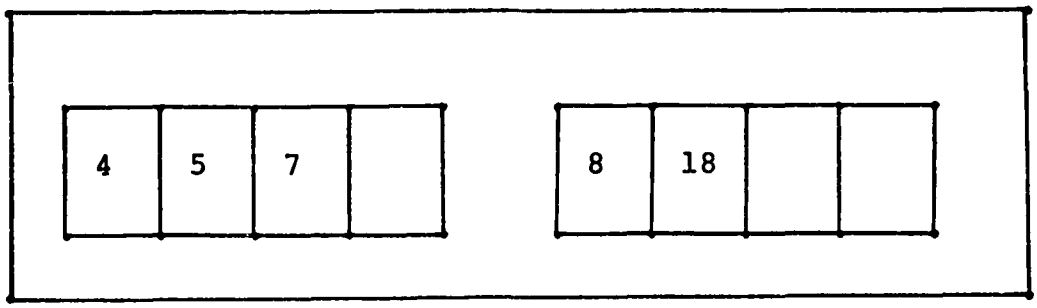


Figure 26. Results of splitting a
a B-tree node

Since the original leaf node has been split, a new leaf node has been created. Consequently, a new maximum value has to be inserted in the tree. The insertinleafnode module indicates the result of the operation performed on the leaf node in a flag called opstatus. The value in this example is 'split' since the leaf node was split. In addition, the maximum value of the new leaf created and the maximum value of the old leaf node are passed up to the insert in B-tree module along with pointers to the new leaf node and the old leaf node.

The next step is to update the internal nodes of the tree. The update process starts with the parent node of the leaf node into which the insertion took place. In Figure 25 this is block 1. First, the status of this node is determined by comparing the number of keys in the B-tree node to the fanout ratio. In this case it is determined that the B-tree node is full. Next, the status of the child node is taken into consideration and the node

is updated depending on the status of the child node. In this case, the status of the child node was 'split' . This indicates that a new value is to be inserted into this node. However, the node is full and thus to perform an insertion, this B-tree node must be split. The results are shown in Figure 27.

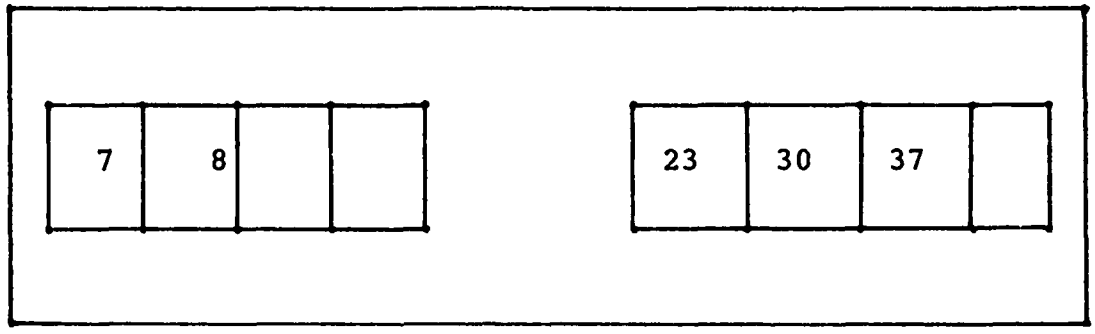


Figure 27. Results of splitting a B-tree node

Now since this was the B-tree root that was split, a new B-tree root has to be created and the maximum values of the nodes split inserted into this new B-tree root. The resulting B-tree is shown in Figure 28.

The Design of the Deletion Modules

Basic Function of the Deletion Operation

The deletion of tuples from a relation file is also a basic edit operation. The function of the deletion process is to determine those tuples which meet the selection criteria, delete these tuples from the appropriate relation, and update the B-trees which

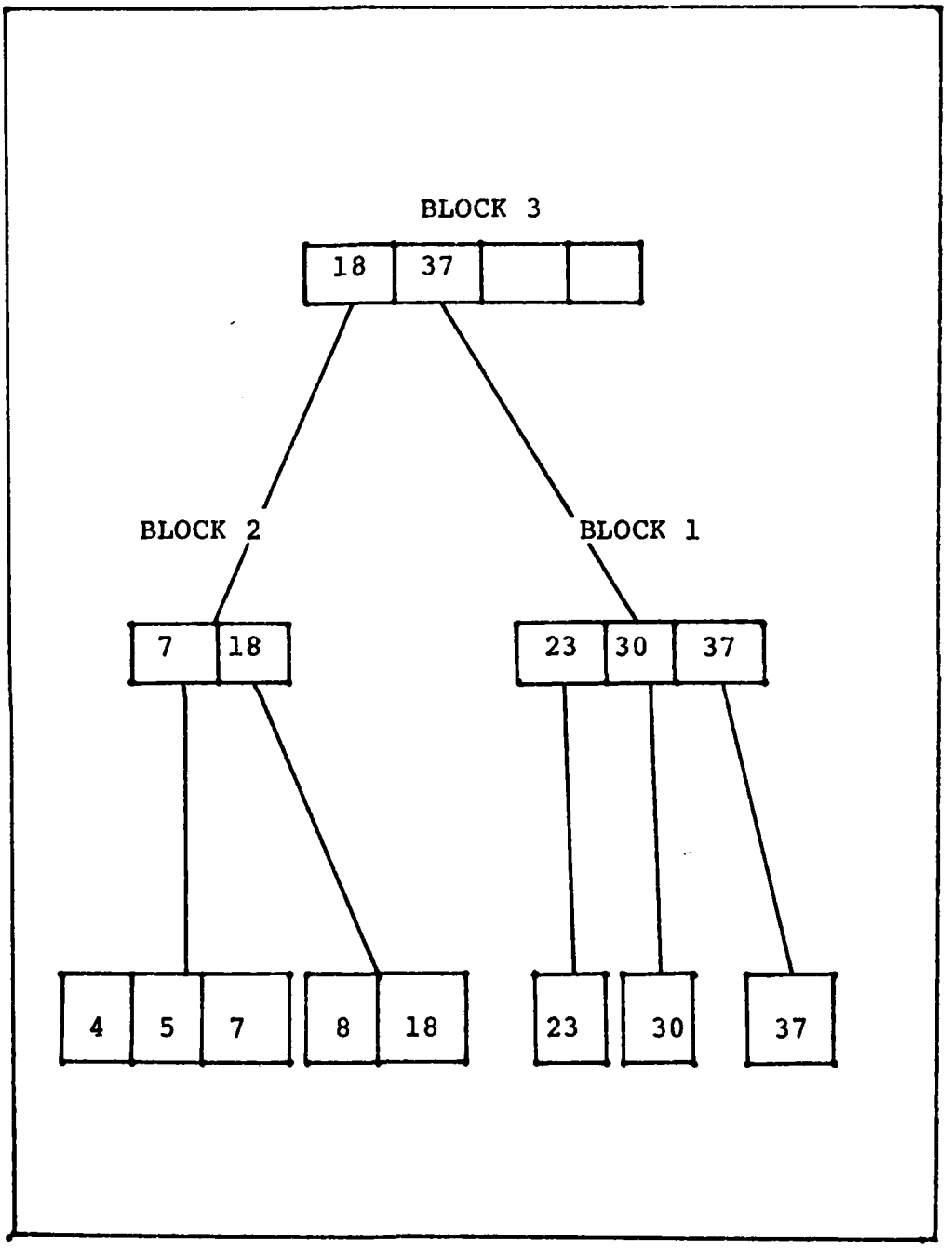


Figure 28. Resultant B-tree

correspond to each attribute of the tuples deleted. This function is not as simple as the insertion operation because some means of determining the tuples to delete must be determined. A means of how to determine the tuples will be discussed in the following section.

In order to perform the deletion process, the editor calls a module called delete. The general algorithm for the deletion process is as follows:

```
DELETE ALGORITHM
Begin
  Find tuples to delete
  For each tuple to be deleted do
    Mark deleted in relation file
  endfor
  For each tuple to be deleted do
    For each tuple attribute do
     Delete from appropriate B-tree
    endfor
  endfor
end
```

The structural diagram is given in Figure 29:

The function of this module is given in the following sections along with some idea of their inputs and outputs. The starred boxes represent modules which have not yet been implemented.

The Findtuples Module

The function of this module is to find the tuples which satisfy the selection criteria specified by the user. The input into this module should consist of the selection criteria entered by the user and the output of this module should be a list of tuple identifiers which will serve as input into the next module. In order to find the tuple identifiers for those tuples to be deleted, the

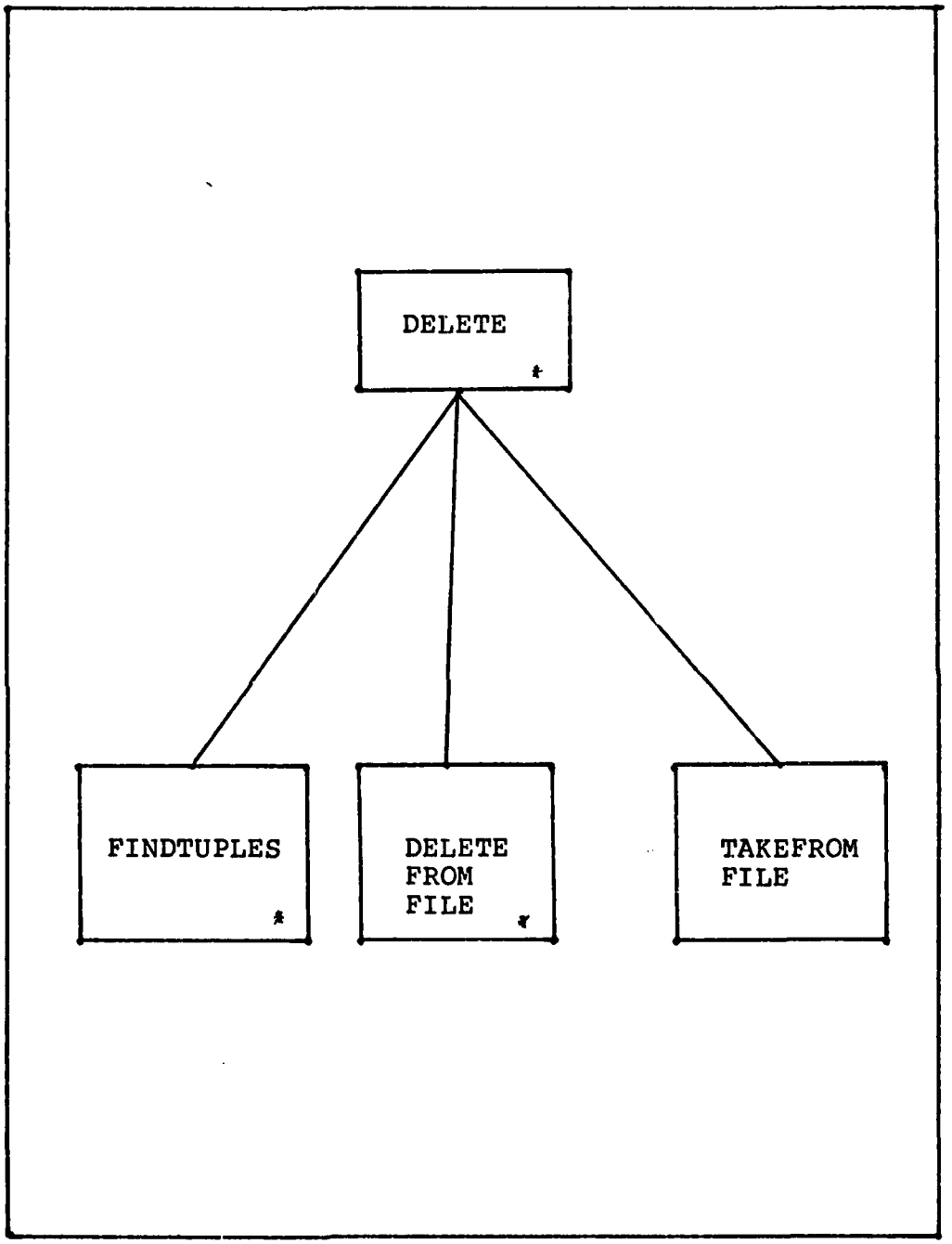


Figure 29. The Delete Modules

modular breakdown is shown in Figure 30.

The Maketidexpression Module

The maketidexpression module takes as input the selection criteria specified. For example, if the user wanted to delete all Air Force and Army officers under the age of 30, the selection expression may be the following:

Branch of Service: =Air Force or =Army and
Age: <30

This module would find all the tuples that satisfied each of the operands. For the example shown, the expression would replace =Air Force with a list of the TID's of all the tuples where the service type was Air Force. It would also replace =Army with a list of the TID's of all the tuples of a specified relation where the service type was Army. Finally, it would replace < 30 with a list of all the TID's that satisfied this criteria. A potential breakdown of this module is given in Figure 31.

Note that in order to find the TID's, the leaf nodes must be searched, and the TID's actually returned would depend on the operator. For instance, if the operator was "greater than", one would find the leaf node that was equal to the value specified using the B-tree. Then a sequential search of all the leaf nodes after this node would be performed. All the TID's would then be returned in these leaf nodes if they were of the correct attribute and

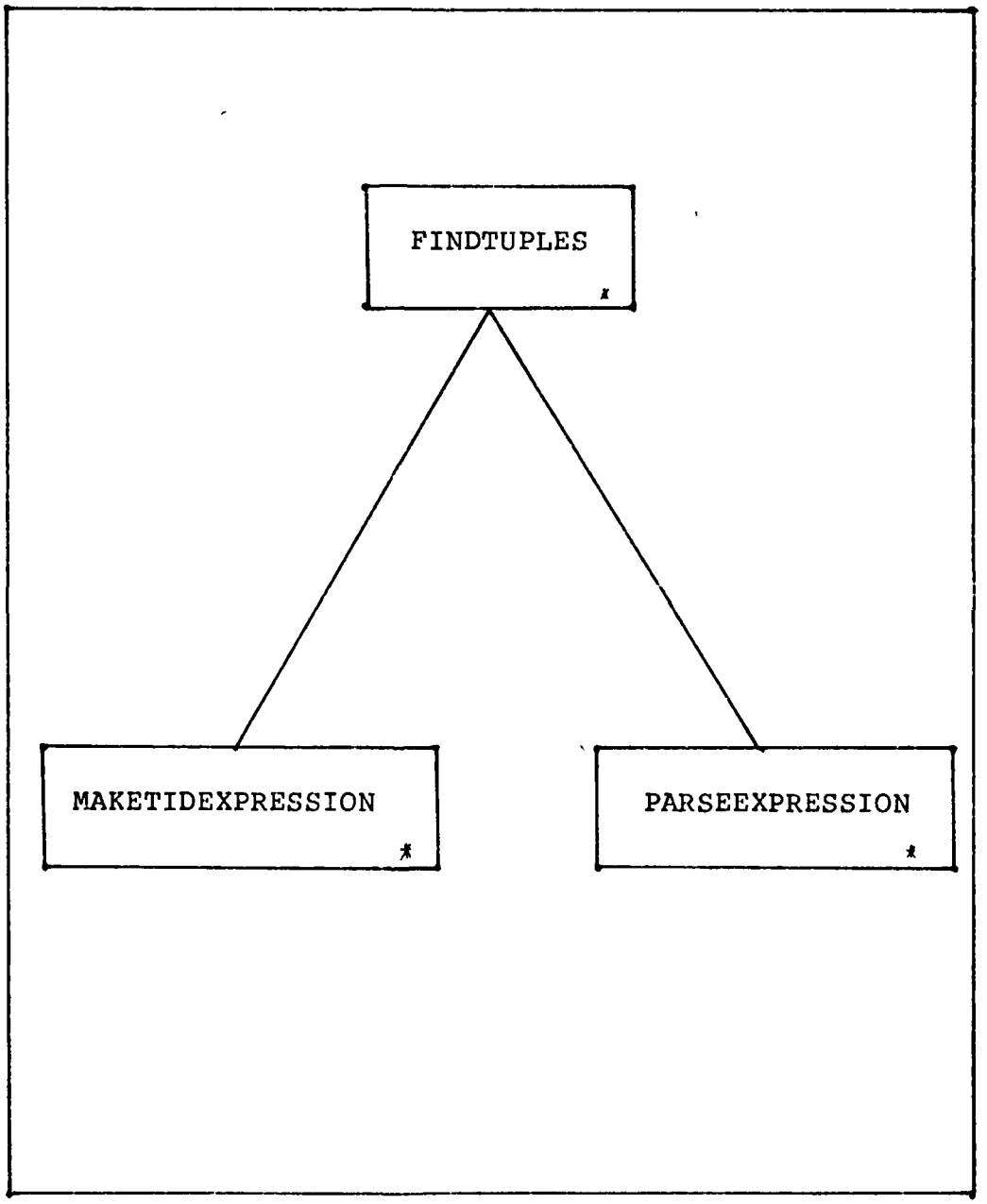
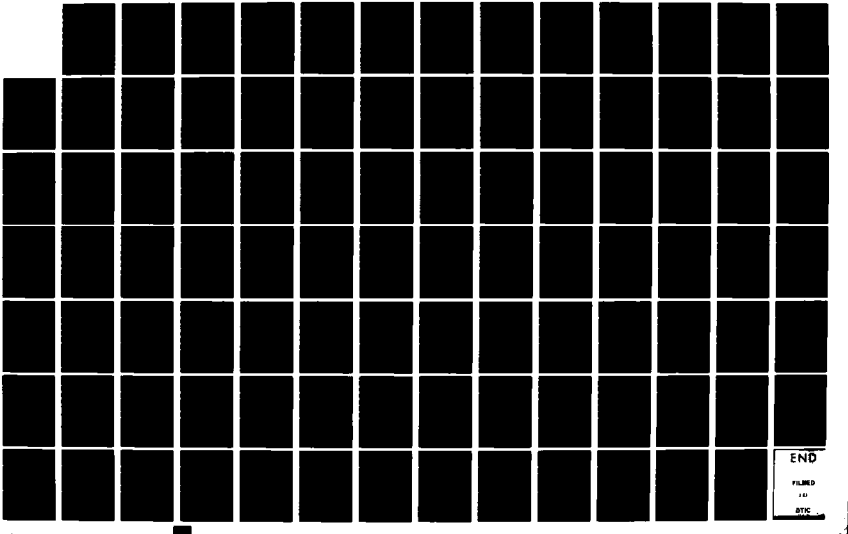
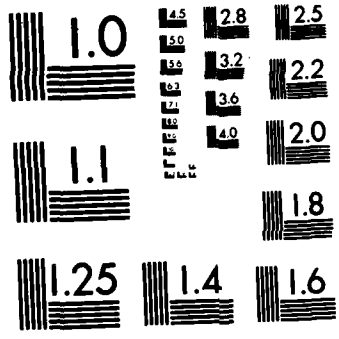


Figure 30. The Findtuples Module

AD-A124 927 THE CONTINUED DESIGN AND IMPLEMENTATION OF A RELATIONAL DATABASE SYSTEM(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGINEERING 2/2

UNCLASSIFIED L M RODGERS DEC 82 AFIT/GCS/EE/82D-29 F/G 9/2 NL





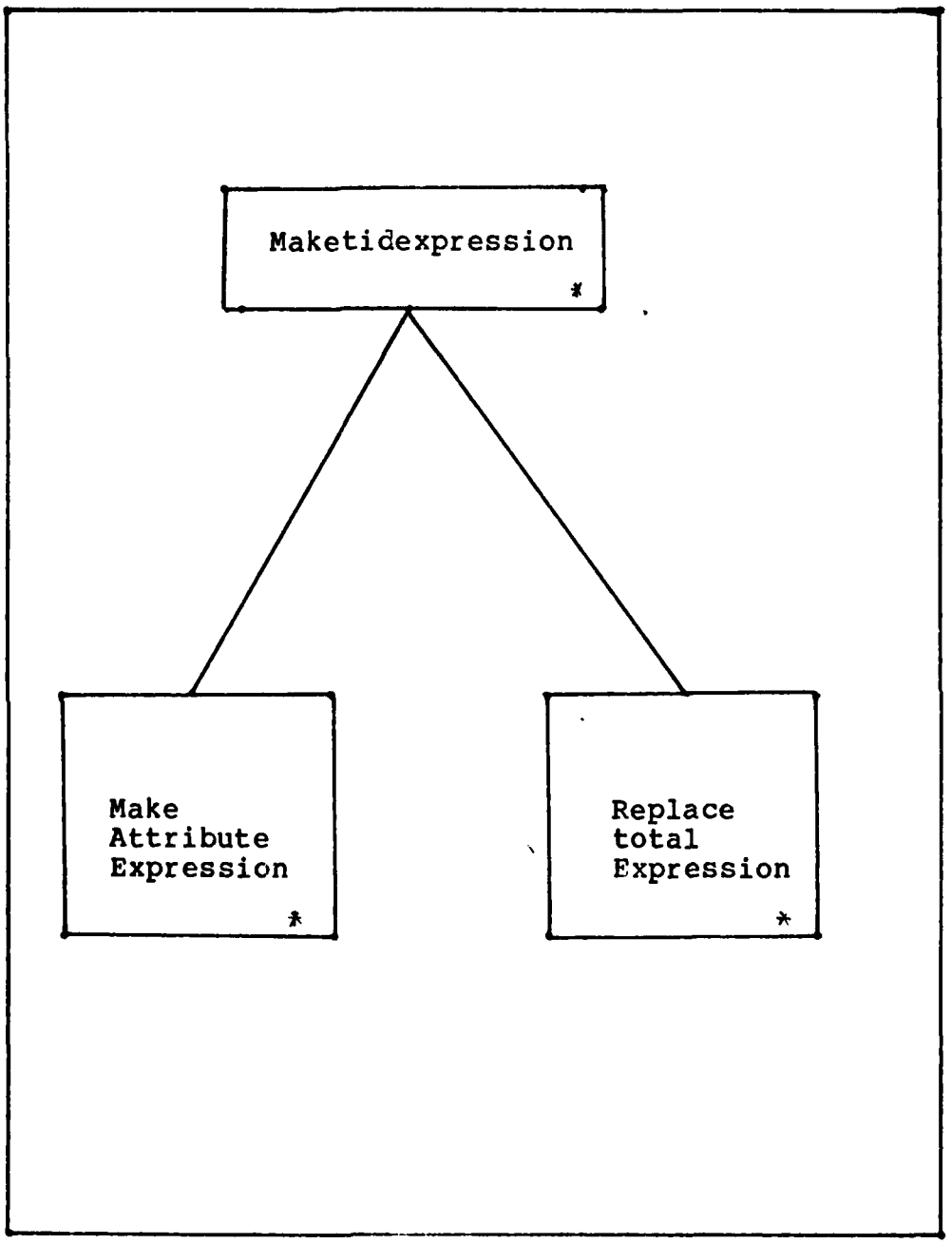


Figure 31. Maketidexpression Module

relation specified.

The Parseexpression Module

The next step is to parse the expression returning a list of TID's which is the result after the TID expression has been evaluated. Suppose for the example given above the results of the maketidexpression module is the following:

Branch of Service:	(Roster 1 1, Roster 1 2, Roster 2 1) or (Roster 3 1, Roster 5 2) and
Age:	(Roster 1 1, Roster 3 1, Roster 1 2, Roster 2 2)

The parseexpression module would evaluate this expression, and return the following TID's:

(Roster 1 1, Roster 1 2, Roster 3 1)

The Deletefromfile Module

The function of this module is to mark as deleted those tuples selected in the appropriate relation file. Input into this module is a list of TID's. Output can be a list of TID's or the list of the actual tuples deleted.

The TakefromBtree Module

The function of the takefrombtree module is to delete an attribute value from the leaf node and update the tree if necessary. The input into this module can either be a list of TID's with the associated relation name or a list of the Tuples themselves. The decision is based on a tradeoff between memory space and processing time. If memory space is a critical resource, it may be best to pass

the TID's to this module. However, this may increase processing time because the tuples would have to be retrieved one at a time from disk. If the tuples are actually retrieved when they are deleted from the relation file, this would save processing time, but may take up vital memory space. It is thought, however, if development is continued on the LSI-11 under the UCSD Pascal System Version II, that the TID's be passed as input to this module, and each tuple be retrieved separately. However, this can only be done if the tuple is marked deleted, but actually removed from the relation file. If this is the case, the modular breakdown is shown in Figure 32.

The general algorithm is given as follows:

```
TAKEFROMBTREE ALGORITHM
For each TID do begin
  Get tuple
  For each attribute in tuple
    delete from leaf node
    if maximum value in leaf node changed or
       leaf node left empty then
       delete value from B-tree
    endif
  endfor
endfor
```

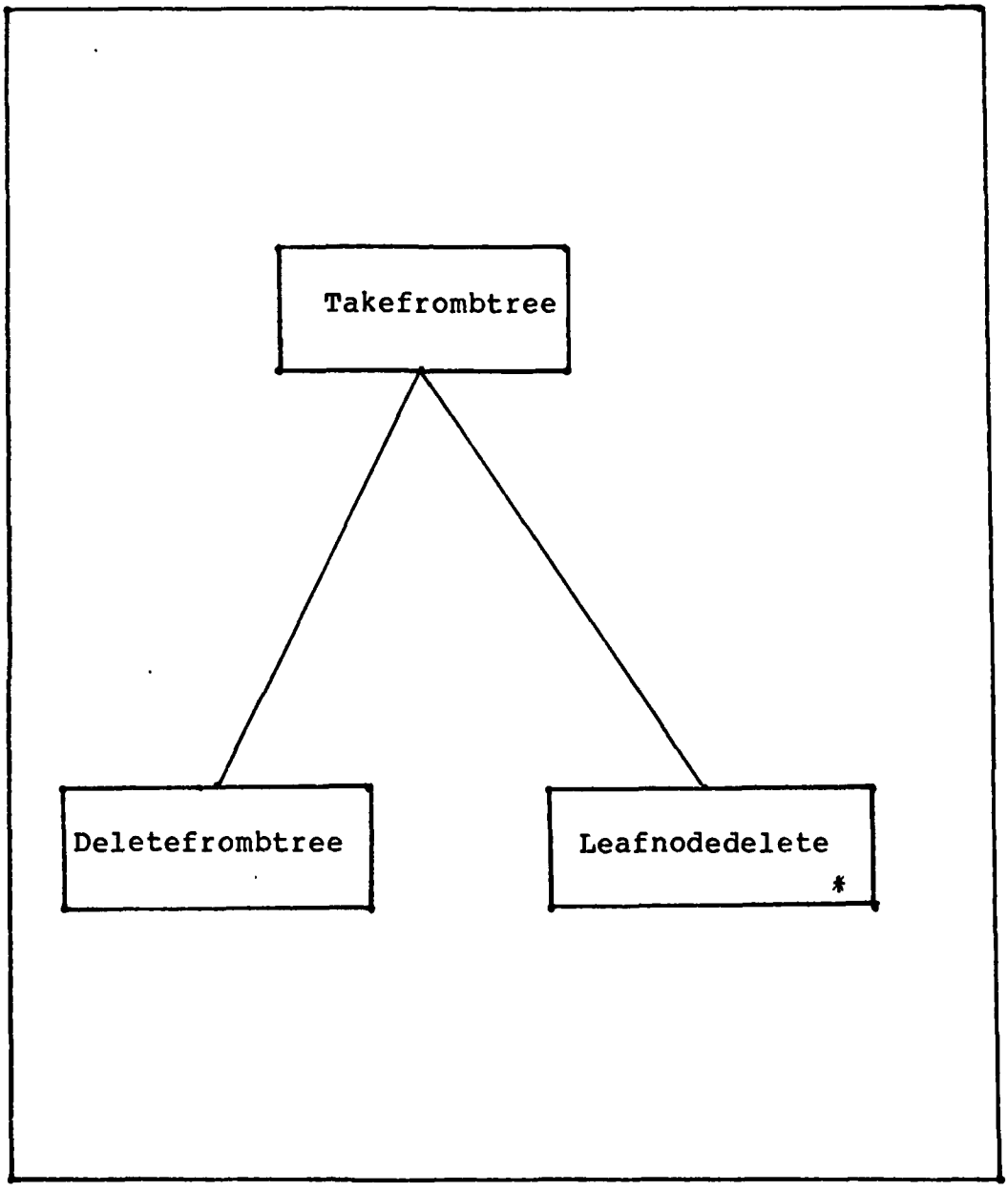


Figure 32. The Takefrombtree Module

The Deletefromleafnode Module

The function of this process is to delete a key value from a leaf node. Essentially, this entails finding the leaf node in which to perform the deletion operation. In order to find the leaf node, the tree file to search has to be determined and the root to the tree has to be found. The root information can be determined from the tree file header. Once the leaf node has been located it must be read from disk and the node must be searched for the key value to be deleted. Once found, it must be marked deleted and the number of keys in the leaf node must be updated if the key value just deleted was the last key of that value. Once this has been accomplished, the status of the leaf node must be returned along with the maximum value of the leaf node before the deletion occurred and the maximum value of the leaf node after the deletion occurred. If the leaf node is left empty, the status value returned is 'empty', otherwise, the value 'alright' is returned. If the status is 'empty' the new maximum value of the leaf node is set to indicate an empty string. In addition the leaf nodes have to be relinked so that the empty node is no longer included in the leaf chain. A summary of the procedure described is given by the following algorithm:

```
DELETEFROMLEAFNODE ALGORITHM

Begin
  If the B-tree exists then
    get the B-tree root
    find the leaf node
    search the leaf node for the value to delete
    delete the value
    return the status of the operation
  Endif
End
```

A general structured diagram is given in Figure 33:

The Deletefrombtree Module

The function of this process is to delete a domain value from the B-tree. Specifically, this module should be called only if the deletion of a key value from a leaf node leaves the leaf node empty or the maximum value in the leaf node is changed as the result of the deletion of a key value. The inputs into this procedure are explained as follows:

DELETEFROMBTREE INPUTS:

OLDMAXVALUE:	the maximum value of the leaf node before the deletion took place
NEWMAXVALUE:	the maximum value of the leaf node after the deletion took place. If the leaf is left empty, this variable should be set to some default value
STATUSOFCHILD:	the status of the child or leaf nodes. Valid values are empty, indicating that the child was left empty, or alright, indicating that the child node was not left empty
PRESENTPTR:	the pointer to the current node. The first call to delete should set this variable to the pointer of the B-tree root.

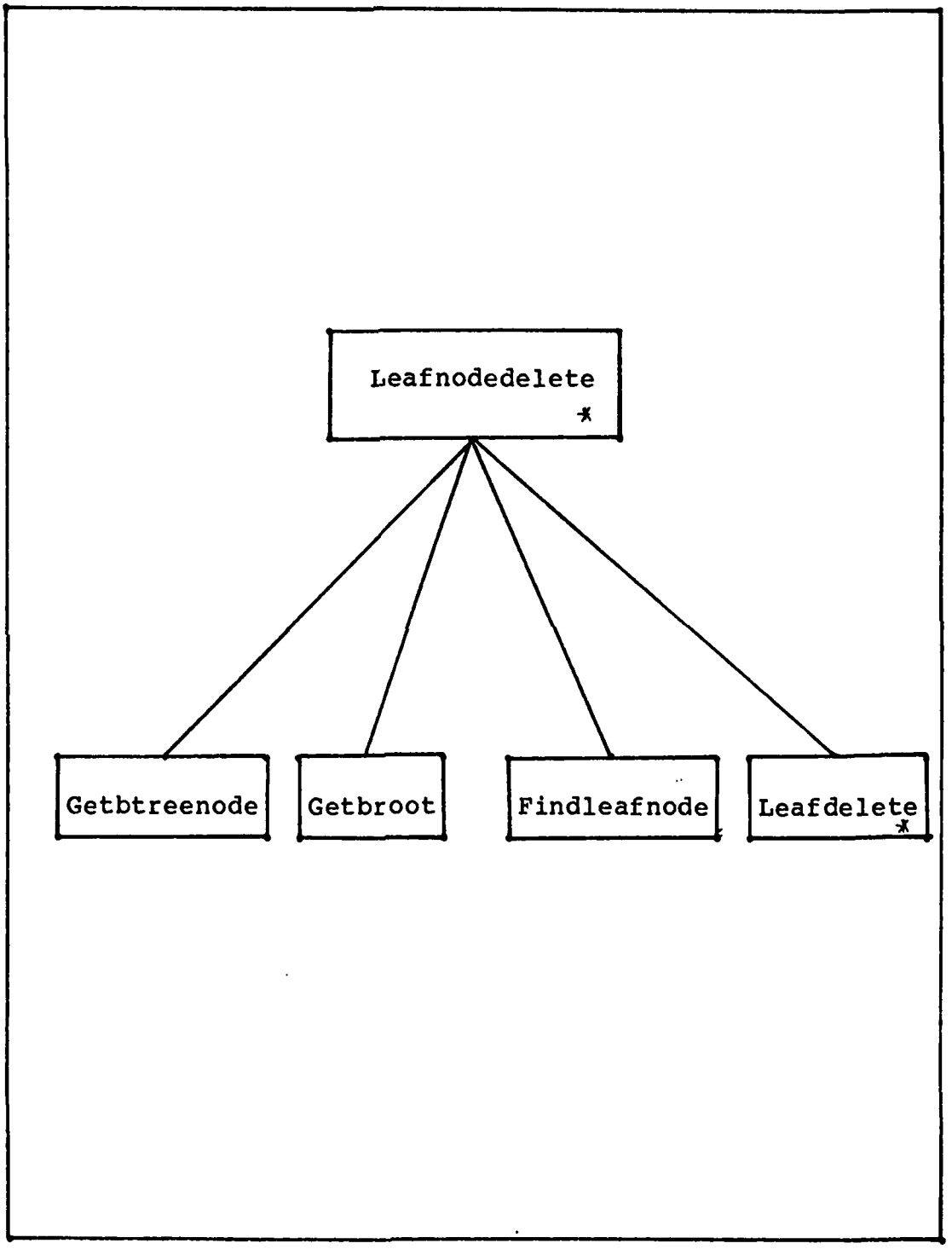


Figure 33. The Leafnodedelete Module

PARENTPTR: pointer to the parent of the present node. The first call should set this variable to the root of the B-tree.

STOPPTR: pointer to the first node in the B-tree index to be updated. This should be the parent of the leaf node in which the deletion was performed.

The General Algorithm for Deletions

The general algorithm for the deletion process is given as follows:

```
DELETEFROMBTREE ALGORITHM

Begin
  If the present node <>  the parent of the
   leaf then
    Get the present btree node
    Search for the next node pointer in the
     present node
    Deletefrombtree
  Endif

  Get the present btree node
  Case status of the child node
    Empty:  Save max value of the child node
            Get the max value of the present node
            Delete max value of the child node
              from the present node
            Get new max value of the present node
            Set status of child to indicate
              operation performed
            Write present B-tree node to disk
            If present node is the B-tree root and
               the number of keys in node = 1 and
               the pointer is not a leaf pointer then
                Collapse the B-tree by one level
    Alright: If the max value in the child node
               changed after the deletion then
               Save old max value of the child node
               Get max value of the present node
               Change the domain value in the present
                 node
             Endif
             Set status of child to indicate
             operation performed
             Write present B-tree node to disk
  Endcase
End
```

Design of the Modification Modules

The function of the modification operation is to determine those tuples to be modified, modify the tuples in the relation file, and update the B-tree of each attribute modified. The process of updating the B-trees is basically seen as a series of calls to the Putinbtree and Takefrombtree modules. Specifically, a high level algorithm for this process is as follows:

```
MODIFY ALGORITHM
Begin
  Findtuples that satisfy modification criteria
  For each tuple to be modified
    Modify tuple in the relation file
    For each attribute modified begin
      get the old attribute value of tuple to
      be modified
      delete this old value from appropriate
      B-tree
      insert new attribute value into B-tree
    endfor
end
```

In order to find the tuples which have to be modified, the same procedures that are developed to find the tuples to be deleted can be used.

PROBLEMS ENCOUNTERED

During the implementation and testing of the tree, several problems were encountered. Specifically, a problem with the stack overflowing was encountered during the compilation of the segment that contains the insertion modules. This was considered to be the result of the stack reaching its limit because of the number of procedures and parameters being passed between the

procedures. To remedy this problem, some parameters were made global and thus could be accessed globally, However, if this is done, care has to be taken concerning how these global variables are changed and their effect on the operation of other procedures.

An additional stack overflow problem was encountered during execution and testing of the code. This was thought to be caused by the fact that primary memory is limited and thus used up during the execution. This is especially a potential problem during the execution of a recursive procedure because the various levels of recursion have to be saved on the stack. A potential solution to this problem is to rewrite recursive procedures in the equivalent straight line code. However, this also may take up an extensive amount of memory when the segment is swapped in. Although this was thought to be an alternative, it was decided that it was not the best solution to consider given the time constraints. Instead, test cases were chosen to verify the procedures without running into the stack overflow problem. It should be noted that this may be a recurring problem, especially since memory is limited and it may be decided to rewrite the recursive procedures if the code can be written efficientl

V. TESTING

Testing is an important phase of any software development cycle. Specifically, testing should be done throughout development. As a consequence, the software developed during this thesis effort did undergo testing. Specifically, it was important to test the algorithms designed for the insertion and deletion into a B-tree because these algorithms were not predefined. In addition, the edit modules were tested.

Test Requirements

In order to test the insertion and deletion algorithms, some criteria for validation needed to be established. It was decided that to ensure some level of validity, it should be required that all routines should be tested. In order to do this, it was decided that all routines written should be invoked at least once. In order to test the edit modules, the same criteria was used.

Design of Test Cases

During the testing phase, it was decided that test cases should be developed to show that each case of the insertion process produced the expected results. In addition, it was decided that test cases should be used to show that the deletion process worked as expected. Consequently, a set of test cases were developed along with drawings of the expected results, i.e. the resultant tree diagrams to ensure that pointers were directed toward

the correct nodes and the domain values were inserted or deleted from the proper records in a node. The insertion and deletion processes were then executed and the results compared to the drawings made. The specific test cases are given in the following sections.

Test Case for Insertion into the B-tree

The test case for the insertion into the B-tree consisted of inserting the following values into a B-tree that was initially empty. The values were inserted in the order presented. They are listed as follows: 18, 23, 30, 37, 5, 45, 2, 55, 11, 15, 65, and 75. The fanout ratio of the B-tree is four. An explanation of how this tests the procedures written is given in the following paragraphs.

The value 18 is the first key value entered in the B-tree. To insert 18 into the tree, the routine "Emptyandchangemax" is called because the leaf node is empty but when the value 18 is entered, the maximum value in the leaf node became 18. Since, the B-tree file was initially empty, the file is initialized by calling the procedure "Initbfile". This entails putting the root node, the fanout ratio, the maximum number of keys in a leaf node and the number of blocks per leaf node in the B-tree file header. The next three values 23, 30, and 37 cause the same routine to be called. This is the routine "Semptyandchangemax". This procedure inserts a new value into a B-tree node if there is room for the value in the node.

The next value, 5, caused a leaf node to split. This resulted in the creation of new leaf node with the maximum value of 7. Consequently, this new maximum value had to be inserted in the B-tree. However, the B-tree node in which the value 7 was to be inserted was full. As a consequence, the procedure "Fsplit" was called since the leaf node was split and the B-tree node to be updated was full. This caused the B-tree node to be split which is accomplished by the procedure "Splitnode". The results of splitting the B-tree node causes the B-tree to increase by one level. This is done by the procedure 'Splitandchangemax'. The B-tree thus far is shown in Figure 34. Fourty-five is the next value inserted in the B-tree. This causes two B-tree nodes to be updated. The first node is block 1 (see Figure 34) which is updated by calling "Semptyandchangemax". The next node is block 3 which is updated by calling "Sfreeandchangemax". This routine is called because the child node had a free space for insertion but after the insertion, the maximum value changed and the present node to be updated had a free space. The resultant tree diagram is shown in Figure 35.

Now suppose the value three is inserted. This causes a change only in the appropriate leaf node. Specifically, the leaf node which begins at block 16 changes.

Next the value 2 was inserted as a test case. As a consequence, the leaf node beginning at block 16 must be

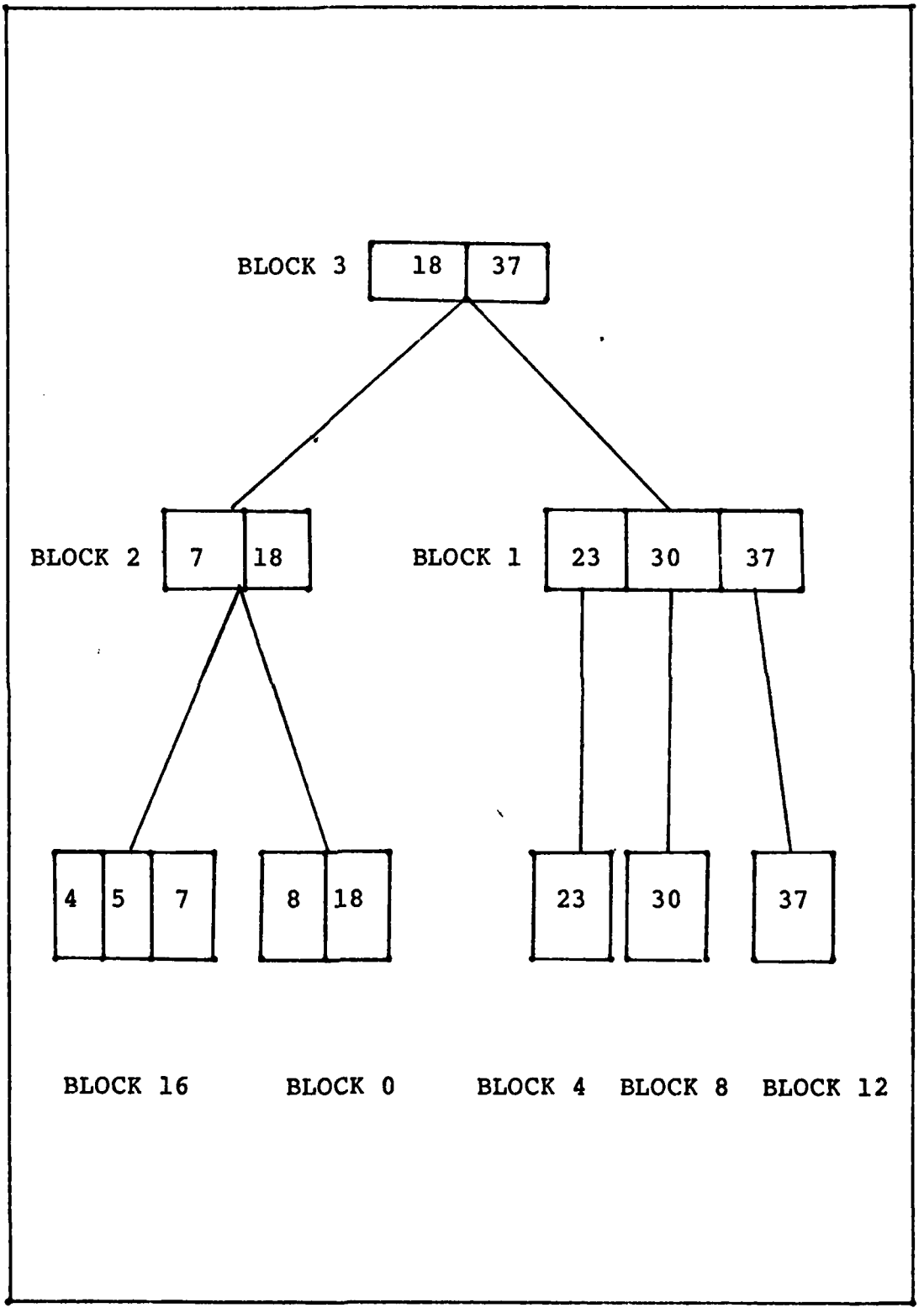


Figure 34. The results of Splitandchangemax

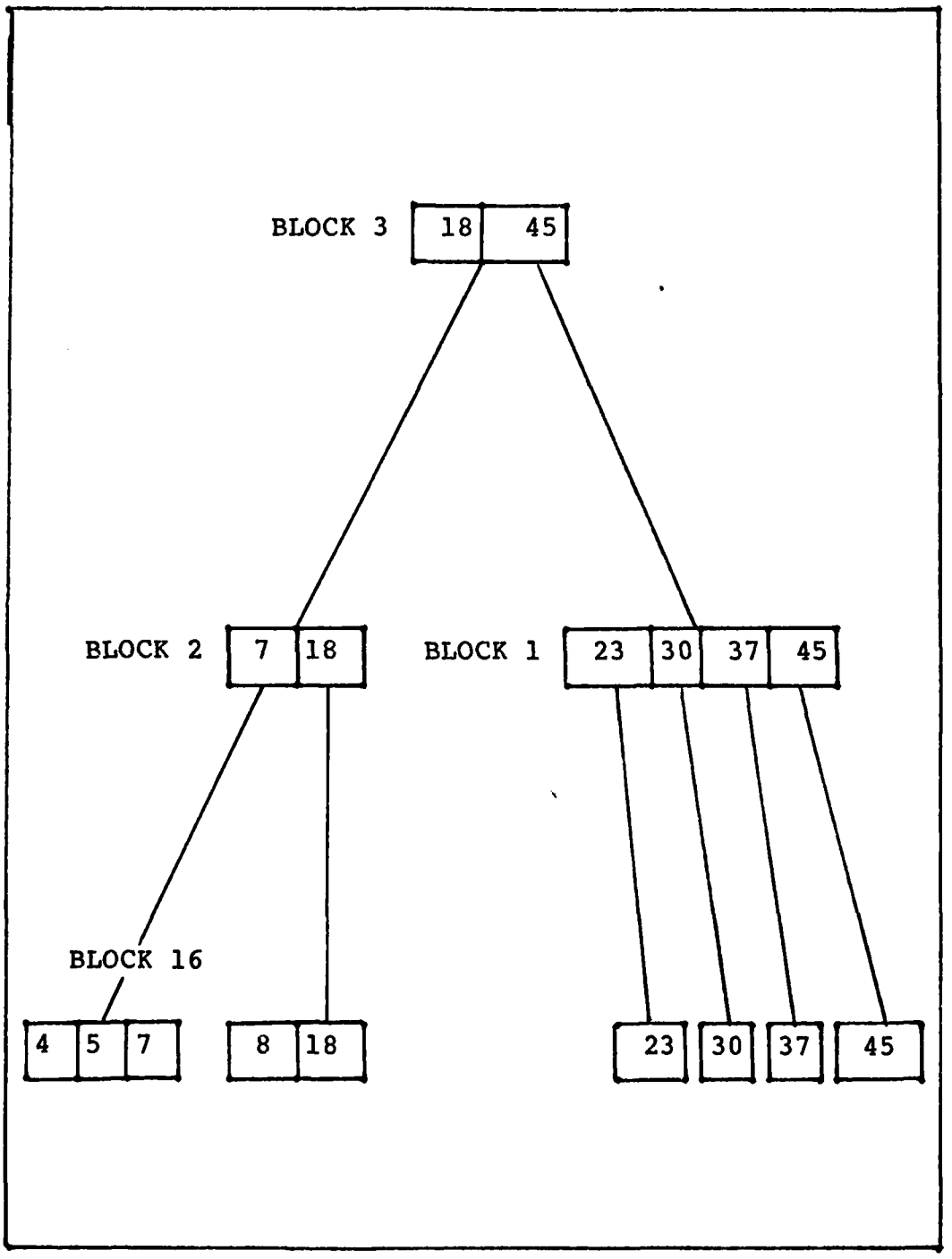


Figure 35. The results of Sfreeandchangemax

split with the result that the maximum value in the newly created leaf node is 4. Since a newleaf was created along with a new maximum value, the B-tree must be updated. To update the B-tree, "Split" is called since the leaf node was split and the B-tree node to be updated has a free space in it.

The next value inserted was 55. This causes two B-tree nodes to be updated. The first node is block 1. To update this node "femptyandchangemax" is called which causes this node to be split. The next node to be updated is the root node. It is updated by calling "Splitandchangemax". The resultant B-tree is shown in Figure 36. The next test case called for the insertion of key value 11. This is accomplished by calling the routine "Ssplit". The resultant B-tree node and child nodes are shown in Figure 37. To continue testing the next value inserted is 15. This causes a split in leaf node zero which results in a new leaf node being created with the maximum value of 14. Consequently, this value has to be entered in the B-tree. The update of the B-tree begins at block 2 and is accomplished by calling procedure "Fsplit". Next the B-tree root has to be updated which is accomplished by "Ssplit".

The next value inserted is 65. This causes block 1 to be updated by calling "Semptyandchangemax". The Btree root is also updated. This is accomplished by the procedure "Ffreeandchangemax".

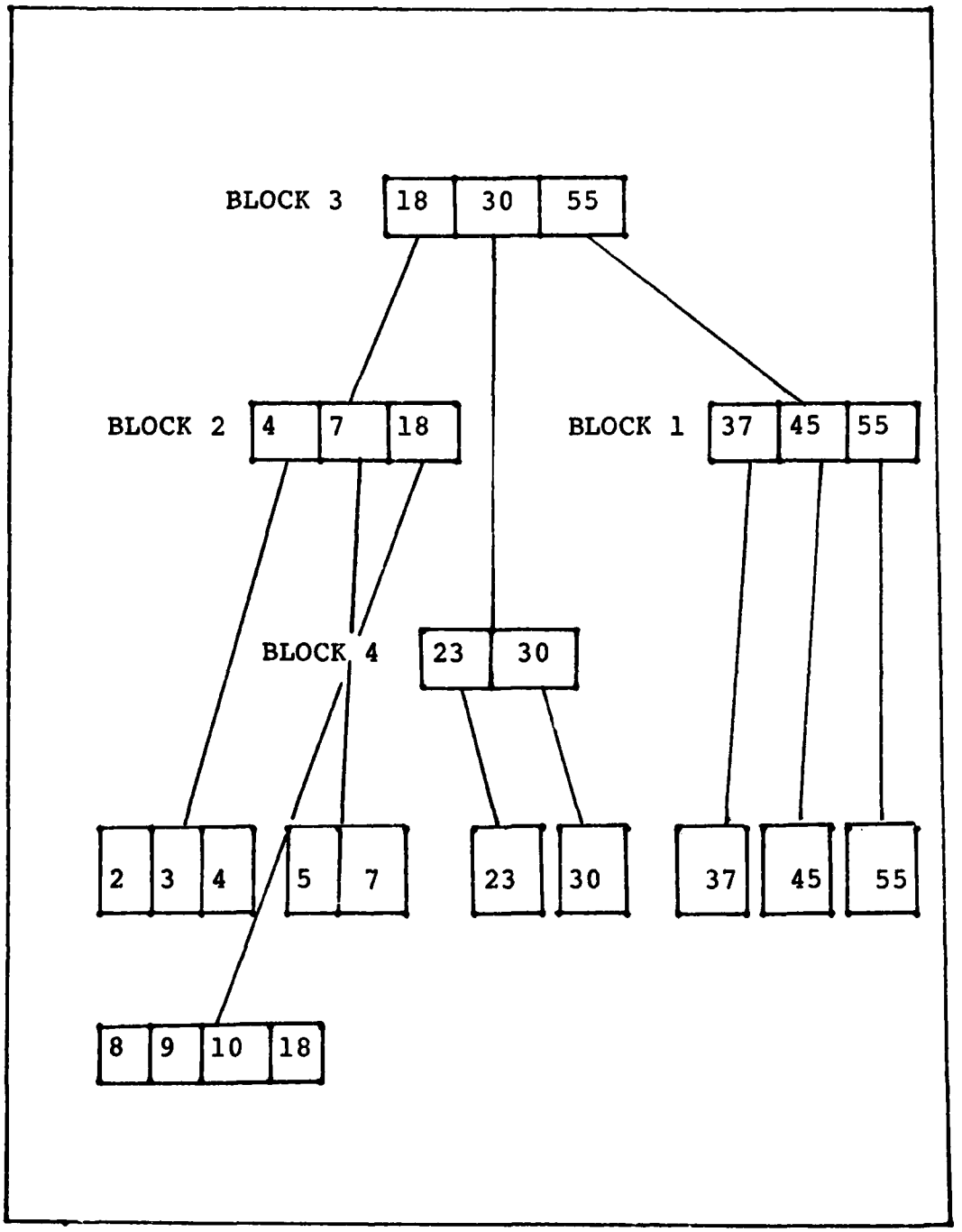


Figure 36. The results of the next call to Splitandchangemax

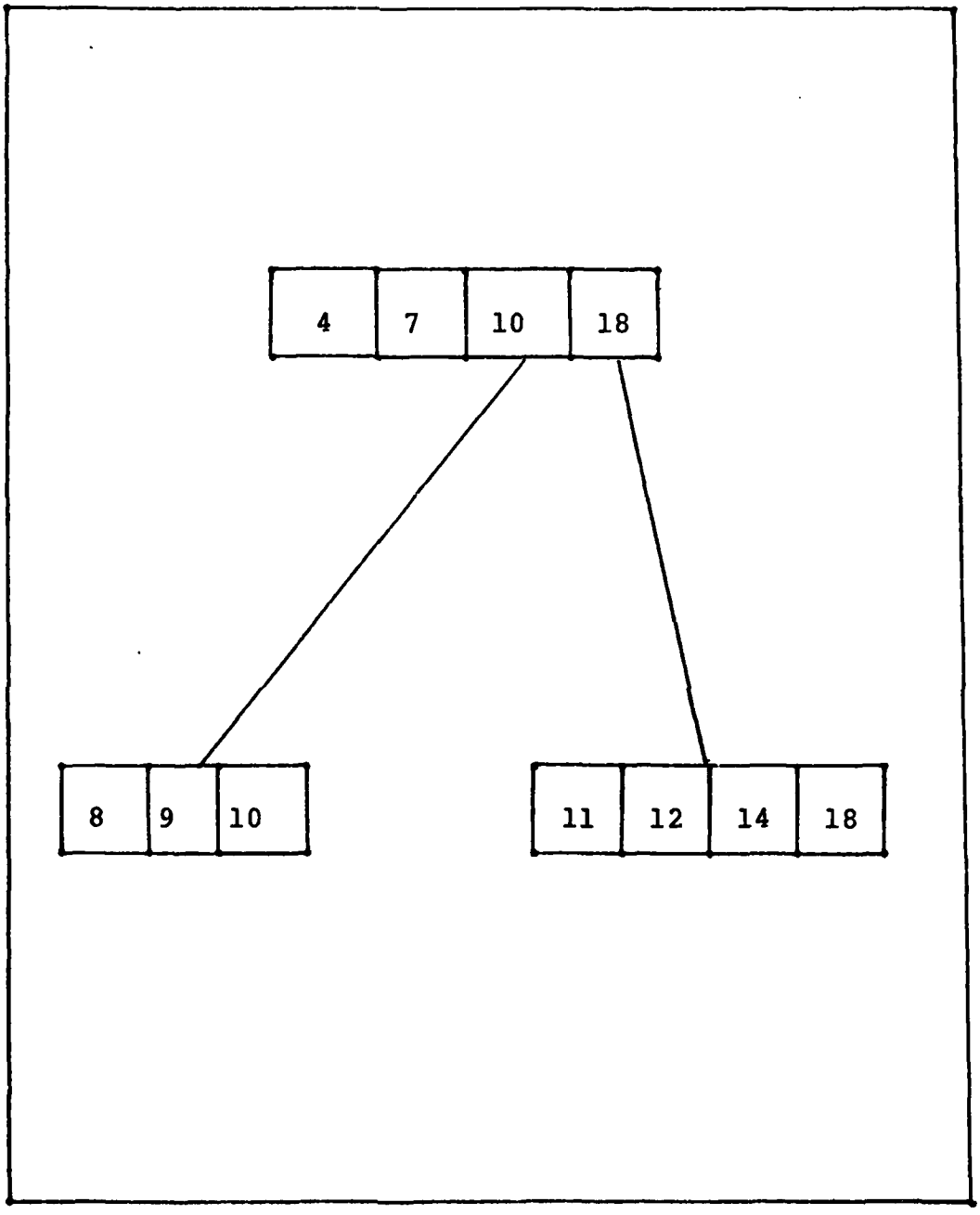


Figure 37. The resultant btree node and leaf nodes

The final value entered is 75. This causes node 1 to be split by calling "Femptyandchangemax". Next, node 3 has to be updated which is accomplished by the procedure "Fsplitandchangemax". The result of this update is the node is split and thus a new B-tree root has to be created. This is accomplished by procedure "Splitandchangemax". The resultant B-tree is given in Figure 38. The leaf nodes shown are just a representation of what would actually take place if this section of code had been implemented at the time this module was tested.

Test Case for Deletion from the B-tree

Several test cases were formulated for the deletion of values from a B-tree. They are given as follows. The first test case began with the B-tree shown in Figure 39.

From this B-tree the following values were deleted: 65, 23, 4 and 7. The resulting B-tree is shown in Figure 40. An additional test case was performed to see if the tree would decrease by one level properly. The initial B-tree is given in Figure 41.

The following values were removed from the B-tree with the resultant shown in Figure 42. The values removed are 23, 30, 37.

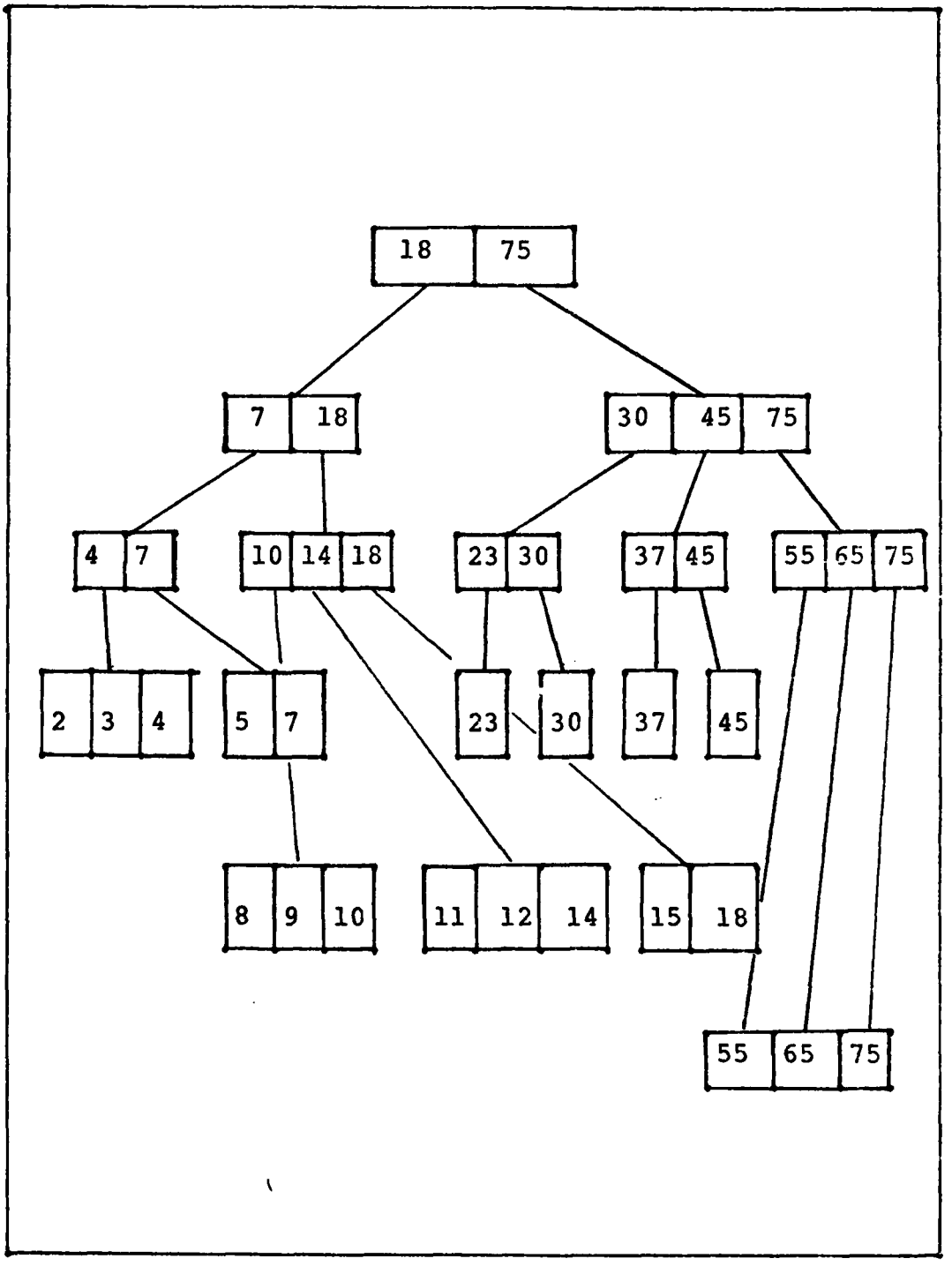


Figure 38. Resultant B-tree for insertion test

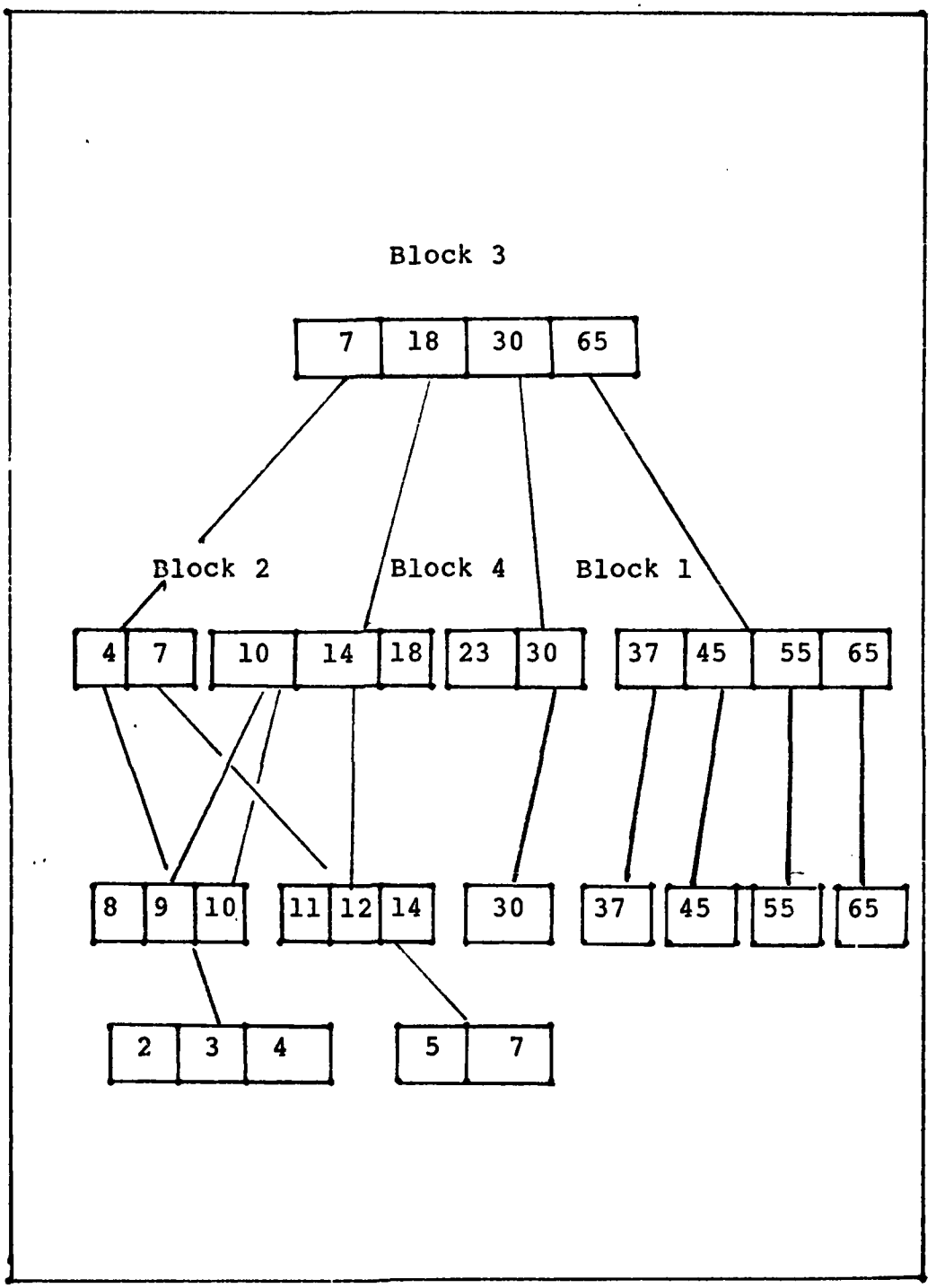


Figure 39. Initial B-tree for Deletion Test

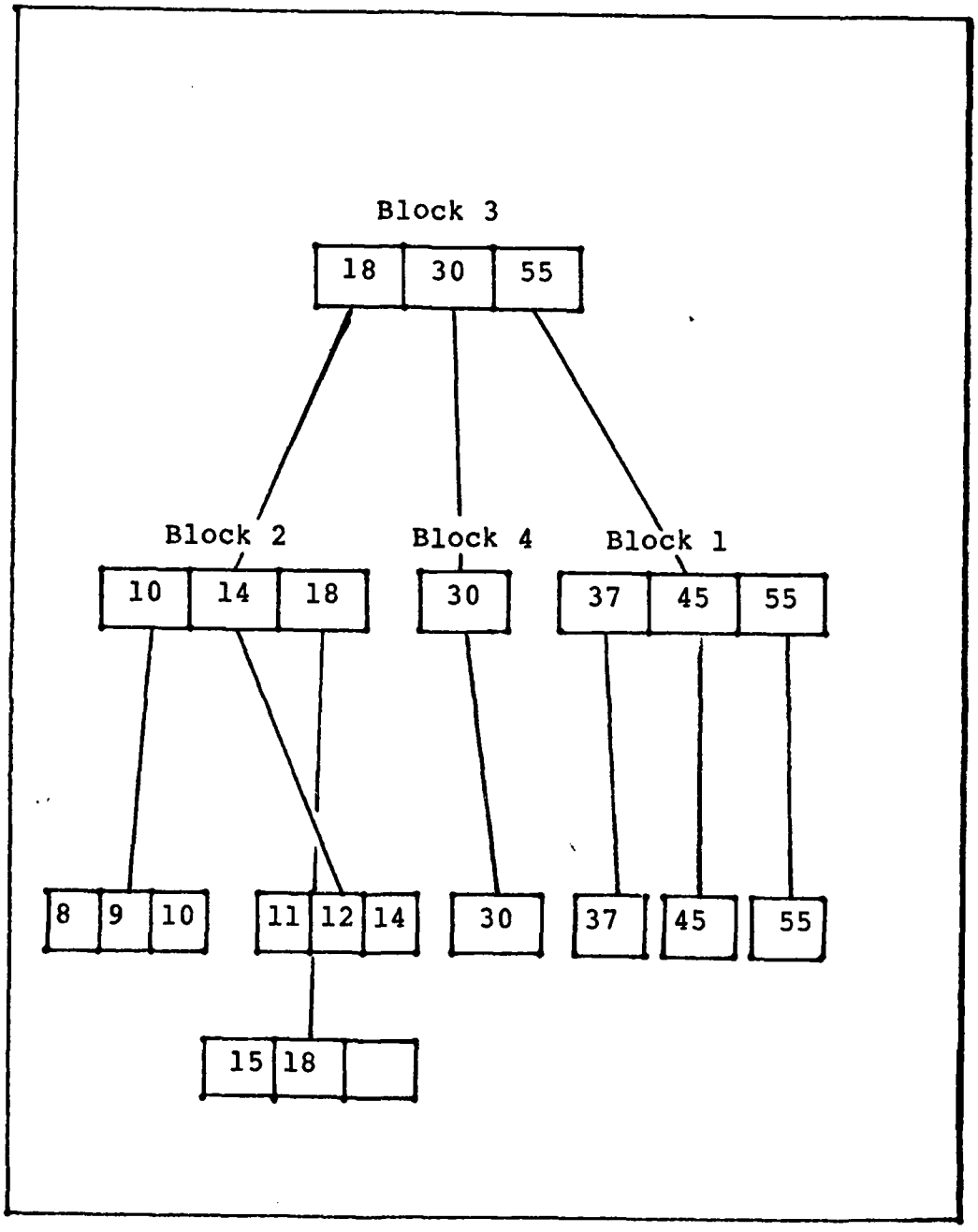


Figure 40. Resultant B-tree for deletion test

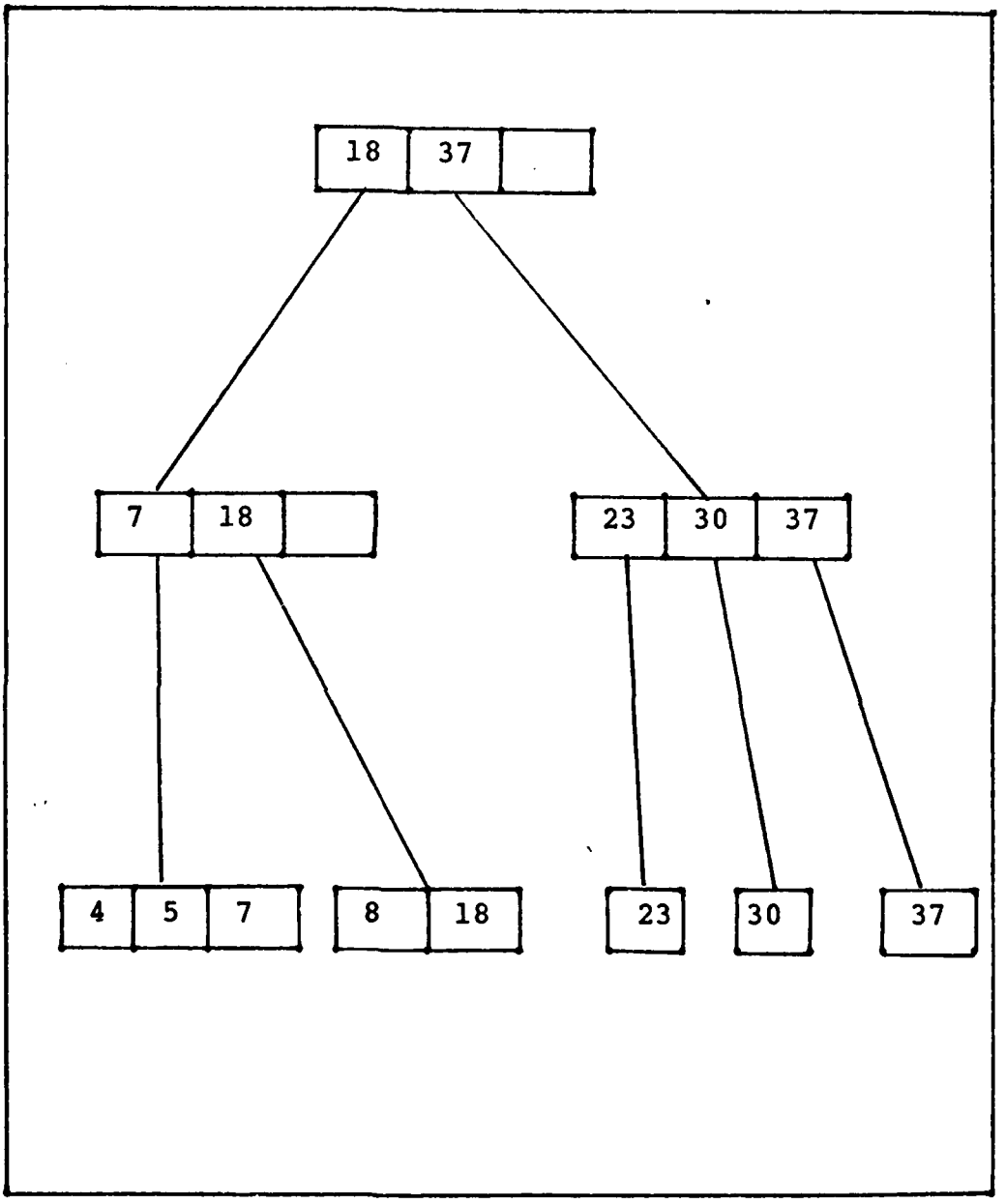


Figure 41. Initial B-tree for testing the collapse of a root node

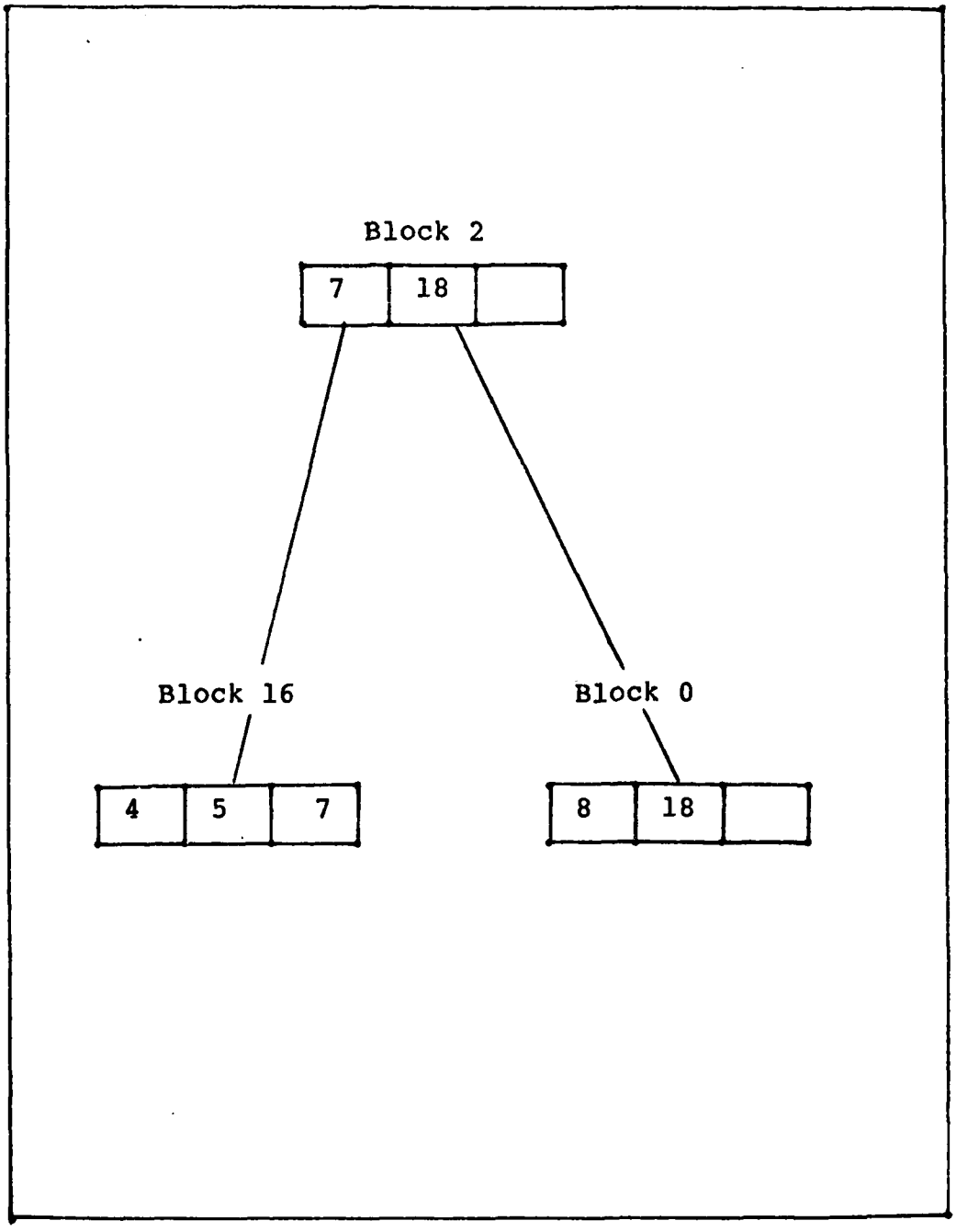


Figure 42. Resultant B-tree after collapse of the root node

VI. CONCLUSIONS

It was stated at the onset of this thesis effort that the goals were to further the design and implementation of the AFIT Relational Database System. This specifically entailed the continued design and implementation of the edit modules and the low level access structure. In order to accomplish these tasks, it was necessary to design algorithms which have the potential to reduce processing time and introduce efficient memory usage. However, during this thesis effort it was found that certain tradeoffs existed between the processing time and efficient memory usage. Specifically, on the LSI- 11, it seems that memory is a critical resource. Consequently, some modules should be implemented so that space could be conserved. As a result, processing time may be increased.

Toward the completion of these goals, the DDL processor was made a separate program, the editor interface with the user was completed and tested, and a design was developed for the low level access mechanism and its interface with the system editor. In addition, algorithms for the insertion into the B-tree and deletion from the B-tree were developed, implemented and tested. From the test cases performed, the algorithms do produce the expected results.

Recommendations

There are several aspects left of the development of

the AFIT Relational Database System which can be considered for future projects and follow on thesis efforts. Specifically, the problem still remains to fully design and implement a completely running relational database system. This requires fully laying out the requirements and specifications of such a system. In addition, a design of how the run modules should interface with the low level access structure needs to be considered and then implemented. Also, the low level access mechanism and the interface with the editor needs to be completed. Specifically, this entails the completion of the modules which perform insertions and deletions into the leaf nodes.

BIBLIOGRAPHY

1. Comer, Douglas. "The Ubiquitous Btree," Computing Surveys, 11: 121-137.

2. Date, C. J. An Introduction to Database Systems. Reading: Addison-Wesley Publishing Company, 1981.

3. Haerder, Theo. "Implementing a Generalized Access Path Structure for a Relational Database System," ACM Transaction on Database Systems, 3: 285-298 (1978).

4. Hardgrave, Terry W. "Ambiguity in Processing Boolean Queries on TDMS Tree Structures: A Study of Four Different Philosophies," Transactions on Software Engineering, 6: 357-372 (1980).

5. Held, Gerald and Michael Stonebraker. "Btrees Re-examined", Communications of the ACM, 21: 140-143 (1978).

6. Horowitz, Ellis and Sahni Sartaj. Fundamentals of Data Stuctures, Maryland: Computer Science Press, Inc. (1976).

7. Kreps, Peter, Michael Stonebraker, and Eugene Wong. "The Design and Implementation of Ingres," ACM Transactions on Database Systems, 1· 189-222 (1976).

8. Lien, Y. E. "Design and Implementation of a Relational Database on a Minicomputer," Department of Computer Science, University of Kansas, Lawrence, Karsas.

9. Mau, James D. "Implementation of a Pedagogical Relational Database System on the LSI-11 Microcomputer," (Thesis) School of Engineering, Air Force Institute of Technology, Wright Patterson AFB, Ohio 1981.

10. McCreight, Edward M. "Pagination of B*-trees with Variable Length Records," Communications of the ACM, 20: 670-674 (1977).

11. McCreight, E. and R. Bayer. "Organization and Maintenance of Large Ordered Indexes." Acta Informatica, 1: 173-189 (1971).

12. Meldman, Jay., et al. Riss: A Relational Database Management System for Minicomputers. New York: Van Nostrand Runhold Company, 1978.

13. Nelson, Dale E. Database Minor Exam. School of Engineering, Air Force Institute of Technology, Wright Patterson AFB, Ohio, 1982.

14. Plieth, Peter G. An Implementation of a Co-ordinating Operator Constructor. School of Engineering, Air Force Institute of Technology. Wright Patterson AFB, Ohio, 1979.

15. Rosenberg, Arnold L. and Lawrence Snyder. "Time and Space

Optimality in B-trees," ACM Transactions on Database Systems, 6: 175-193 (1981).

16. Roth, Mark A. The Design and Implementation of a Pedagogical Relational Database System. (Thesis) School of Engineering, Air Force Institute of Technology, Wright Patterson AFB, Ohio, 1979.

17. Schneiderman, Ben. Databases: Improving Usability and Responsiveness. New York: Academic Press, 1978.

18. Smith, Miles and Philip Yen-Tang Chang. "Optimizing the Performance of a Relational Algebra Database Interface." Communications of the ACM, 18: 568-579 (1975).

19. Wiederhold, Gio. Database Design. New York: McGraw-Hill Book Company (1977).

20. Yao, S. Beng, and David Dejong. "Evaluation of Database Access Paths". Proceedings of the International Conference on Management of Data. New York: Association for Computing Machinery, Inc (1978).

AFIT/GCS/EE/82D

USERS'S GUIDE: THE AFIT RELATIONAL DATABASE SYSTEM THE DATA DEFINITION FACILITY

APPENDIX A

TO

THE CONTINUED DESIGN AND IMPLEMENTATION
OF THE
AFIT RELATIONAL DATABASE SYSTEM
THESIS

BY

LINDA M. RODGERS
2LT USAF

Graduate Computer Systems
December 1982

CONTENTS

APPENDIX A

USER'S GUIDE: THE AFIT RELATIONAL DATABASE SYSTEM

THE DATA DEFINITION FACILITY

INTRODUCTION AND OVERVIEW * SECTION 1

The Data Definition Facility (DDL Facility) described in this document is a program intended to run on a stand alone minicomputer under the control of the UCSD Pascal Operating System, Version II.O. All software is written in Pascal, resulting in relatively straightforward maintenance and enhancement.

This program is designed to be used primarily with a CRT terminal as the CONSOLE device.

The function of the DDL Facility is to provide the Data Base Administrator with the capability to define domain and relation definitions. Originally this code was part of the host database system of which original development was done on an Intel 8080 micro-processor with dual drive floppy disks and an ADM-3A CRT terminal and later all code was transferred to the LSI-11 (Ref 9). The code which allowed the user to define domains and relations has been separated into the DDL Facility. This program only allows the DBA to construct domain and relation definitions. In addition, the DDL Facility allows the DBA to obtain a full listing of the domain and relation definitions. Also, the DBA is given the capability to destroy these definitions.

1.1 The Data Definition Facility: An Overview

The structure of the data definition facility is best conceptualized in terms of the "tree- like" structure diagram in Figure A-1.

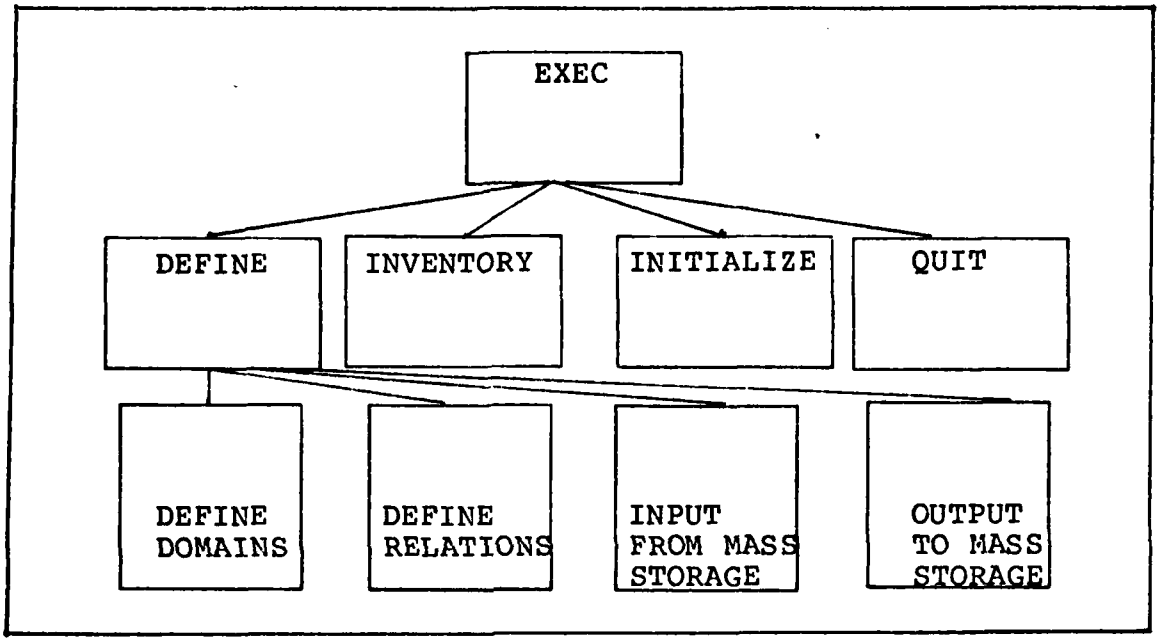


Figure A-1. The Data Definition Facility

While the DBA is in a particular level, the facility displays a list of available commands called the "prompt-line." The prompt- line will usually appear at the top of the screen. Commands are usually invoked by typing a single character from the CONSOLE device. For example, the prompt line for the outermost level of the facility is:

```
Sys Options: D(efine), I(nventory), Z(Initialize), Q(uit)
>
```

By typing "D" the user will "descend" a level within

the structure diagram into a level called "DEFINE". Upon entering DEFINE, another prompt-line detailing the set of commands available at the DEFINE level of the facility is displayed. The Q(uit) command causes the user to exit from DEFINE and "ascend" back to the outermost command level of the program. Now the user is back at the level in the program from which he started. Some commands within the program prompt the user for more information, such as the name of a relation, a line number, etc. In these cases, the user enters the required information followed by a carriage return (<cr>). If an error is made in typing a portion of the information, the backspace key, CTRL H, or equivalent key depending upon the system configuration may be used to "back over" and erase the erroneous part. If the user decides not to enter any information at all, escape from most commands is by not entering any characters, ie. type <cr>. Unless otherwise stated, any input to a yes/no question besides a "Y" is considered a no.

FORMAT, KEY WORDS AND NAMES * SECTION 2

As stated previously, the DDL Facility allows the DBA to define domain and relation definitions. The format for the domain and relation definitions are predefined by the system software. They are given as follows. A domain definition consists of the following information:

```
NAME = domain name
TYPE = CHAR/TEXT/INTEGER/REAL
N,M  = maximum number of characters in a value of a
       particular domain
```

where N is the following:

-- the maximum number of characters if type is CHAR or TEXT

-- the maximum number of digits if type is integer

-- the maximum number of digits before the decimal point if type is real

and where M is the following:

-- the number of digits after the decimal point if the type is real.

The format for the relation definition is the following:

```
NAME       = relation name
ATTRIBUTES = list of attributes
     NAME        = attribute name
     DOMAIN      = domain name
     CONSTRAINTS = list of constraints
     SORTED UP/DOWN/NOT SORTED
```

where CONSTRAINTS has the following format:

```
OPERAND = L(ist) of values
          G(reater than)
          S(less than)
          E(qual)
VALUE or LIST of VALUES
```

It should be mentioned that when defining domain and relation definitions the user is prompted for the appropriate information. All information entered by the DBA except for the relation name may be any combination of 1 to 80 nonblank characters. The relation name may be any combination of 1 to 15 alphanumeric characters. In addition, the characters ".", "-", "_", and "/" are allowed. User defined names should not be any of the

following keywords:

ALL	DIVIDE	JOIN	PRODUCT	UNION
AVG	FROM	MAX	PROJECT	VETO
BY	GIVING	MIN	SELECT	WHERE
COUNT	IN	ONE	SORT	$ $
DIFFERENCE	INTERSECT	OVER	SUM	

HOW TO USE THE DDL FACILITY * SECTION 3

3.1 Starting the System

To start the system, execute the code file DDL.CODE. The disk which contains this file must remain on- line during the execution of the system in order to permit segment swapping. However, the disk may be in any drive.

3.2 The Logon Procedure

The DDL Facility is only to be used by the DBA. This is to allow centralized control over the construction of domain and relation definitions. To insure that only the DBA has access, a special identification name is assigned to the DBA. The indentification name is bb$$RROOTTHH. The "b" represents a blank and must be entered as part of the identification name. In order to change the identification name, the constant DBMID which is declared in the COMMON UNIT must be changed.

To logon to the DDL Program, a set procedure must be followed. The program indicates that it has been properly initiated by welcoming the user to the AFIT RELATIONAL DATABASE SYSTEM and displaying its version number. Next, the user is prompted for his identification number as follows:

Enter Identification -->

Here, the DBA should enter the special identification name that differentiates him from other users. If the correct id is entered, the program responds by welcoming the DBA to the system. Otherwise, the program indicates to the user that entrance has been denied, and promptly terminates.

Once the DBA has properly logged on, he is then asked if the system has 1 or 2 disk drives. If 2 drives are indicated, then the domain and relation definitions are assumed to exist on the disk drive in DRIVE 1, in a file called SETUP.DATA. Otherwise DRIVE 0 is assumed to contain the diskette with SETUP.DATA. If SETUP.DATA cannot be found, the DBA is asked if he would like one created by the following message:

```
Can't find SETUP.DATA on the disk in Drive 0/1.
Should I create one? --(Y/N)-->
```

Here, the DBA should answer with a Y or N. If the DBA indicates that he would like to create a file called SETUP.DATA, the program then writes a file called SETUP.DATA to DRIVE 1, if there are two disk drives. If only one disk drive, SETUP.DATA is written to DRIVE 0. If the DBA should decide not to create a new SETUP.DATA, the program will respond with the following prompt:

```
Insert proper disk in DRIVE 1 & hit return -->
```

The program will respond with this message until

SETUP.DATA can be properly found or until the program has been halted by hitting the break key. Once SETUP.DATA is found, the outer level prompt line is displayed as follows:

Sys Options: D(efine), I(nventory), Z(Initialize), Q(uit)

Each option is described in detail in the following section.

3.3 Outermost Level Commands

A. I(nventory)

To obtain a listing of all domain and relation definitions type "I" which will start the execution of the Inventory procedure. The Inventory procedure allows a list of all the domain and relation definitions currently defined to be printed on the CRT. The format for the domain listing is the following:

```
                List of DOMAINS defined:

        NAME = domain name
        TYPE = Char/Text/Integer/Real (N,M)
        .
        .
        .
```

where N is the following:

- -- the maximum number of characters if type char or text
- -- the maximum number of digits if type is integer
- -- the maximum number of digits before decimal point if type is real

and where M is the following:

- -- the maximum number of digits after decimal point if type is real.

The format for the relation definitions is the following:

List of Relations defined:

Name = Relation Name/No relations are currently defined

Attributes are:
  Name = attribute name
  Domain = domain name
  Sorted UP/Sorted DOWN/Not sorted
Constraints = op value/No constraints

Keys are:

  Key attribute name
  Security is
    ID = identification name assigned
    READ = READ ID
    INSERT = INSERTION ID
    DELETE = DELETE ID
    MODIFY = MODIFICATION ID

Existence is permanent/temporary
No. of tuples = tuple number
Stored in file = filename

To allow the user to take his time in viewing the domain and relation definitions, the user is given some control over the automatic scrolling capability. Specifically, when the screen becomes full, the continued display of the definition is halted. The following message is printed on the screen:

*** TYPE Return to continue **

If the user enters a (<cr>), display of the domain or relation definition is continued until the screen again becomes full or all the definitions have been viewed. If the screen again becomes full, the message is printed on the CRT. This process continues until all domain and relation definitions have been displayed.

B. Z(Initialize)

To initialize the database system, the DBA must type a "Z". This destroys all domain and relation definitions. Once this process has been completed the following message is displayed:

Initialization Complete

C. D(efine)

To define relation or domain definitions type a "D" at the outermost level and the following prompt will be displayed:

```
1)  Define Domains
2)  Define a New Relation
3)  Input from Mass Storage
4)  Output to Mass Storage
5)  Quit

Select 1 - 5 -->
```

The individual define commands are invoked by typing the number to the left of the parenthesis.

3.4 Innermost Define Commands

A. Define Domains

To construct a domain definition enter a "1" and the user is then prompted for the domain name as follows:

```
Enter DOMAIN name --->
```

Here, the DBA should enter a combination of 1 - 80 nonblank characters. Once a domain name has been entered the following prompt is displayed:

```
Enter DOMAIN Type: C(haracter), T(ext), R(eal), I(nteger)-->
```

Here, the user should enter a "C", "T", "R", or an

"I" depending on the type of domain he wishes to have defined. If an R is entered, the user is then asked to enter the maximum number of digits before the decimal point and the maximum number of digits after the decimal point as follows:

```
Enter max no. of places before decimal point-->
Enter max no. of places after decimal point -->
```

The user should respond by entering integer values. Otherwise, a zero will be assumed. If the domain type entered is other than a real number, the user is prompted as follows for the maximum number of characters or digits allowed.

```
Enter max no. of characters or digits allowed-->
```

If the domain has previously been defined, an appropriate message is displayed and the user is asked the following question:

```
Do you still want to define this domain? -(Y/N)->
```

If the answer is yes, the process of prompting the user for the domain name and type is repeated.

If the domain is properly defined, the user is next asked if he would like to define another domain. If the user desires to define another domain, the process is repeated. Otherwise, the user is no longer prompted for information and the process of defining domains is terminated.

B. Define a New Relation

Defining a new relation is a more complex process than defining a domain. To construct a relation definition, the user is first prompted for a relation name as follows:

Enter relation name --->

Here, the user should enter a combination of 1 to 15 characters. Next, the user is asked for a list of attribute and the corresponding domain name as shown:

```
Enter attribute name, return, and domain name
for this attribute, after each prompt (>).
Type Return alone to quit attribute definitions.
```

An example of how the user should respond is given as follows:

```
>ATTRIBUTE NAME (<cr>)
 DOMAIN NAME (<cr>)
>(<cr>)
```

Next, the user is asked if a directory should be maintained for the attribute and if so, whether or not it should be sorted in ascending order or descending order. To indicate that the directory should be sorted in ascending order, the user should enter a "U". If the directory is to be sorted in descending order, the user should enter a "D". Note that specifying that a directory should be maintained is optional. However, if a directory is maintained, querying processing time may be greatly decreased. Next, the user is asked to enter the number of attributes that will serve as a key. Then the user is prompted for each attribute name that is the key value. If

the attribute entered has not been defined the user is given an appropriate error message and is prompted again for the attribute name. In addition, a message is displayed if the attribute entered has already been defined as a key.

Once all the key attributes have been defined, the user is then prompted for the following security identification:

```
USER ID
READ ID
INSERT ID
DELETE ID
MODIFY ID
```

The user must enter a user identification number, however, the id's are optional.

Finally, the user is asked if constraints should be made for an attribute by the following:

```
Do you want any constraints? --(Y/N)-->
Constraints on what attribute?
```

The user has the following options if constraints are desired.

```
Enter type:  E(equals), S(less than), G(reater than),
             or L for a list of values --->
```

Here, the user indicates his preference by entering either an "E", "S", "G" or "L". After an option is chosen, the program responds with the following prompt if the "L" option is not chosen:

```
Enter constraining value --->
```

Here the user should enter an appropriate value depending on the domain type. If a list of constraints is

specified the system responds as shown:

```
Enter a value at each prompt.
Type RETURN alone if no more.
```

Here the user is prompted for each individual value. Values entered should adhere to domain constraints specified. Note that specifying constraints is optional. In addition, more than one constraint can be specified for an attribute. If so, all values entered for that attribute must meet at least one of the constraints specified.

Once the desired constraints have been specified, the relation is checked to determine if it has been previously defined. If so, the user is prompted for a new relation name. Otherwise the process of defining a new relation is complete.

COMMAND SUMMARY * SECTION 4

4.1 D(efine)

A. Define Domains

Define domains is used to construct a domain definition (refer to section 2). The user is prompted to enter the domain name, the type of domain, ie. Character, Integer, Text, or Real, and the number of characters or digits to be allowed. The domain name must be unique. If other than digits are entered when required then 0 is assumed; and if the maximum integer size of the machine is exceeded, then the maximum integer is used. All yes/no questions must be answered with a "Y" or "N".

B. Define a New Relation

This command is used to define a new relation to consist of the following parts each prompted for individually:

Relation name -- must be unique and no longer than 15 characters.

Attribute name/domain name pairs -- attribute names must be unique within a relation and the domain names must be previously defined. The user is also asked if a sorted directory is to be maintained for each attribute.

Primary key(s) -- the number of keys is entered, and then each attribute which is to serve as a key is entered. The attribute must have been defined and if a duplicate is entered, the user is allowed to prematurely quit specifying keys.

Security passwords -- the DBA enters security passwords for the following:

USERID -- the identification name of the user who defined the relation.

READ -- prevents display of relational operations on the relation.

DELETE -- prevents deletion of any or all tuples.

MODIFY -- prevents modification of any tuple.

INSERT -- prevents insertion of any new tuple.

Constraints - the user is allowed to specify constraints on the attributes of this relation. These are used to further restrict the domain of an attribute. The constraints can be of the form:

$$\text{attribute} \begin{cases} > \\ < \\ = \end{cases} \text{value}$$

or

attribute = (value1, value2, value3,...,valueN).

Multiple constraints can exist on an attribute.

C. Input from Mass Storage

This command has not yet been implemented. Its purpose is to read in data from a disk file into a relation, specifying the format of the data. Data which is incompatible with the domain type or constraints of the relation attributes should be flagged as an error, and non unique key values should also be flagged.

D. Output to Mass Storage

This command has not yet been implemented. Its purpose is to write data in a relation to disk or to a printer in a particular format.

4.2 I(nventory)

This command causes a list of the domains which have been defined and a list of the relations which have been defined and attached to be displayed on the CRT. Note that since this is the DBA routine, all relations are attached and therefore displayed.

4.3 Z(Initialize)

This command allows the DBA to initialize the system variables. This consists of returning all storage and setting the domain and relation definition lists to empty.

CHANGING THE SYSTEM * 5

5.1 Overview

Please refer to sections 3.3.1 and 3.3.2 of the UCSD

Pascal Version II.0 reference manual and current program listings before trying to change the system.

The program is currently divided into a main body segment, two segment procedures and a unit. Each segment is contained in a dummy program in order to permit seperate compilation. The unit contains all types, variables, and procedures which are global to more than one segment. Thus by including the unit in each dummy program, access is allowed to those elements. The format for each segment procedure is:

```
PROGRAM dummy-name;
USES COMMON;  (*the unit is named COMMON*)
SEGMENT PROCEDURE name(parameter-list);
  Local types, variable, and procedures;
BEGIN
  body of name;
END; (*name*)
BEGIN
END. (*dummy name*)
```

Since the segments are separately compiled, the parameter list of the segment procedure must contain all global variables accessed or modified by the procedure. The program or segment procedure which calls each segment procedure must have a dummy segment procedure with the same name, so that it may compile properly. The format for the main body segment is:

```
PROGRAM main;
USES COMMON;
 local labels,types,constants, and variables;
SEGMENT PROCEDURE name1( --- );
BEGIN
END; (*name1*)
SEGMENT PROCEDURE name2( --- );
BEGIN
END; (*name2*)
```

```
.
.
.
other local procedures;
BEGIN
  body of main;
END. (*main*)
```

The format for segment procedures which use other segments is the same as the previous format for a segment procedure except dummy segment procedures are included as local procedures.

Each segment procedure has a particular segment number from 11 to 15 associated with it. The main body segment has number 1 and the unit has number 10. Other numbers are for Pascal use only. The way numbers are assigned to the segment procedures is in first compiled, first numbered order. Thus, in the above format for the main body segment name1 would be assigned 11, name2 assigned 12, etc. Therefore in each dummy program used to define a segment procedure, an appropriate number of dummy segments must exist before the defined segment to ensure that the segment count is the same. Thus, for example, the format for segment name2 would be:

```
PROGRAM dummy name;
USES COMMON;
SEGMENT PROCEDURE dummy name1;
BEGIN
END; (* dummy name1 *)
SEGMENT PROCEDURE name2( --- );
  local labels, types, etc.
BEGIN
  body of name2
END; (*name2*)
BEGIN
END.  (*dummy name*)
```

The unit -- COMMON -- is compiled and placed in the

system library using the librarian program, LIBRARY.CODE. (See section 4.2 of the UCSD Pascal manual.) When each program is compiled, the unit is retrieved from the system library and used in the program. However, each program must still be linked with the system library in order to bind the external variable and procedure references into the unit. After each program is compiled and linked, then the librarian may be used to put all the segments together into one code file. Each code file containing the segment is retrieved and linked into the proper space using its assigned segment number into the overall file.

If a particular segment, including the main body segment, is to be changed then the steps to be followed are:

1) Change the source code for the segment.
2) Compile the program containing the segment.
3) Link the code file to the system library.
4) Using the librarian create a new overall file passing all unchanged segments to the new file and linking in the changed segment.

If the unit has to be changed then after compiling it and placing it into the system library, each program must be recompiled, linked to the system library, and then put together with the librarian.

5.2 Bringing Up the Data Definition Facility

To bring up the Data Definition Facility the following steps should be followed:

1. First, compile COMMON.TEXT such that the object code resides in COMMON.CODE.

2. Execute the System Librarian by typing "X" at the system level. The following prompt line will appear on the CRT:

   EXECUTE WHAT CODE FILE -->

3. Enter "LIBRARY".

4. Next, the following prompt will appear:

   OUTPUT CODE FILE --->

5. Enter a "*" or "SYSTEM.LIBRARY".
6. Next, the user will be prompted as follows:
   LINK CODE FILE --->

7. Enter COMMON.CODE and the screen will appear as follows:

```
LINK CODE FILE ---> COMMON.CODE


0-        0  4-        0  8-        0 12-        0
1-        0  5-        0  9-        0 13-        0
2-        0  6-        0 10-COMMON     14-        0
3-        0  7-        0 11-        0 15-        0

OUTPUT CODE FILE ---> *
```

The user may now link the COMMON segment into segment 10 of the output code file by typing a "10", (<cr>) and "10". In addition to linking this segment into the SYSTEM.LIBRARY, which resides on the system disk, the two segments PASCALIO and DECOPS must also be linked back into the System library, SYSTEM.LIBRARY. In order to do this, type "N" for new. This indicates to the librarian that segments from a new code file are to be put into the output code file. The librarian will respond with the following prompt

LINK CODE FILE --->

8. Enter a "*" or "SYSTEM.LIBRARY".

9. To link PASCALIO, type a "2", followed by a (<cr>), then a "2".

10. To link DECOPS, type a "3", followed by a (<cr>), then a "3".

Now, the SYSTEM.LIBRARY should look like the following:

```
0-              0 4-       0  8-          0 12-        0
1-              0 5-       0  9-          0 13-        0
2-PASCLIO    1824 6-       0 10-COMMON      14-        0
3-DECOPS     2092 7-       0 11-          - 15-        0
```

To exit from the Librarian do the following:

11. Type "Q" for QUIT.

12. ENTER (<cr>) to exit LIBRARIAN.

At this point, the DDL software can be compiled. This software resides in the following text files:

DBMGR.TEXT - the executive for the DDL Facility.

DBDUM1.TEXT - the file that contains the source code for the Define segment.

DBDUM2.TEXT - the file that contains the source code for the Inventory segment.

QUIT.TEXT - the file that contains the source code for writing domain and relation definitions to disk.

To link the DDL source code together, continue the following steps:

13. Compile DBMGR.TEXT making sure that QUIT.TEXT resides on the same disk. Next, compile DBDUM1.TEXT, and DBDUM2.TEXT.

14. Once all three source files have been compiled properly into three appropriate CODE files, they can be linked together into one CODE file

using the LIBRARIAN. To invoke the LIBRARIAN, repeat steps 2 and 3. Once, the prompt has been displayed, enter a valid CODE file name.

15. Next, the following prompt will be displayed:
LINK CODE FILE --->

16. Enter code file for DBDUM1. To link the Define segment into the output code file link segment 11 to segment 11. Next, enter a "N" for new. This will indicate to the LIBRARIAN the fact that segments from a new code file are to be linked into the output file. For the new code file enter DBDUM2.CODE. To link the Inventory segment into the output code file link segment 12 to segment 12. The final code file to be linked is the code file for DBMGR. Type "N" and enter the code file name for the DBMGR object code to link the DB segment into the output code file, link segment 1 to segment 1.

17. To finish the linking process, PASCALIO, DECOPS, and COMMON must all be linked into this output code file. To do this type "N" and for the LINK CODE FILE enter "*" or SYSTEM.LIBRARY. To link these segments into the output code file, link segment 2 to segment 2, segment 3 to segment 3, and segment 10 to segment 10. The output code file should look like the following:

```
0-              0 4-  0  8-           0 12-Inventory
1-DB              5-  0  9-           0 13-          0
2-PASCALIO   1824 6-  0 10-COMMON       14-          0
3-DECOPS     2092 7-  0 11-DEFINE       15-          0
```

18. The final step is to link the output code file by executing the LINKER. To do so, type an "L" at the Pascal system level. The following prompts will be displayed:

Host file? enter output code file that contains all segments to be linked.

Lib file? enter (<cr>).
Map file? enter any valid file name.
Output file? enter DDL.CODE, this will be the executable code file.

To execute DDL.CODE see section 3.1.

AFIT/GCS/EE/82D

USER'S GUIDE: THE AFIT RELATIONAL DATABASE SYSTEM

APPENDIX B

TO
THE CONTINUED DESIGN AND IMPLEMENTATION
OF THE
AFIT RELATIONAL DATABASE SYSTEM
THESIS

by

Linda M. Rodgers
2Lt USAF

Graduate Computer Systems
December 1982

CONTENTS

APPENDIX B

USER'S GUIDE: THE AFIT RELATIONAL DATABASE SYSTEM

INTRODUCTION AND OVERVIEW * SECTION 1

The database system described in this document is a system intended to run on the LSI-11 under the control of the UCSD Pascal operating system, Version II.0. All system software is written in Pascal, which should aid in the maintenance and modification of the system software. The system is designed to be used with a CRT terminal for display purposes.

1.1 The Database System: An Overview

The structure of the system is best conceptualized in terms of the tree-like structure in Figure B-1.

While a user is in a particular level, the system displays a list of available commands called the "prompt-line" at the top of the screen. Commands are usually invoked by typing a single character from the keyboard. For example, the prompt line for the outmost level of the system is:

```
Sys Options: E(dit),R(etrieve),I(nventory),A(ttach),Q(uit)
>
```

By typing "R" the user will "descend" a level within the structure diagram into a level called "RETRIEVE". Upon entering RETRIEVE, another prompt-line detailing the set of commands available at the RETRIEVE level of the system is displayed. The Q(uit) command causes the user

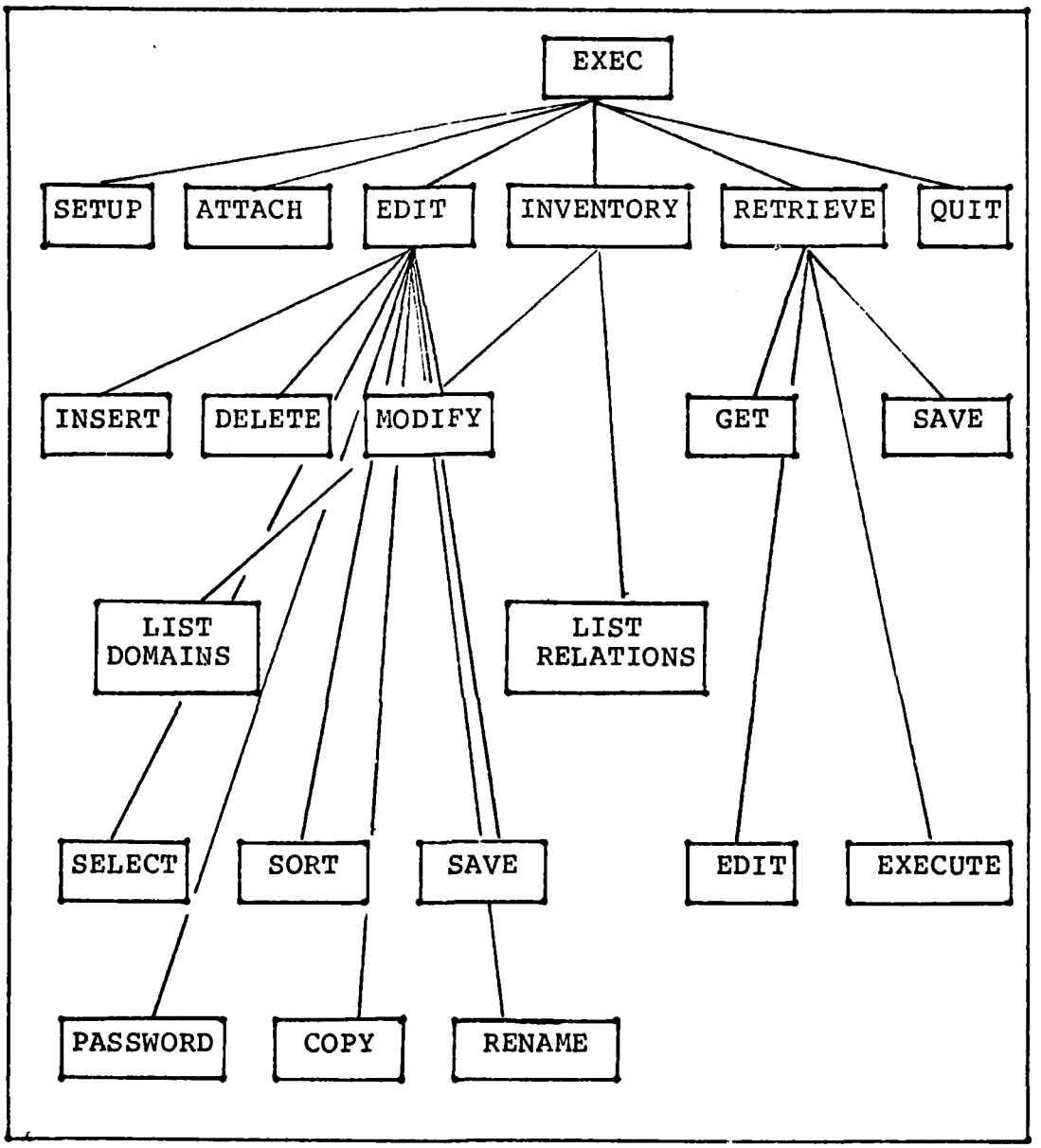


Figure B-1. Roth's System without the DDL

to exit from RETRIEVE and "ascend" back to the outmost command level of the system. Now the user is back at the level in the system from which he started after initially executing the system. Here the user can either choose another option or he can choose to exit from the system by entering a "Q" for quit.

FORMAT, KEYWORDS AND NAMES * SECTION 2

In order to use the database system, it is a good idea to become familiar with the structure of relation and domain definitions. The format for the domain and relation definitions are predefined by the system software. They are given as follows. A domain definition consists of the following information:

```
NAME = domain name
TYPE = CHAR/TEXT/INTEGER/REAL
N, M = maximum number of characters in a value of a
       particular domain
```

where N is the following:

-- the maximum number of characters if type is CHAR or TEXT

-- the maximum number of digits if type is Integer

-- the maximum number of digits before the decimal point if the domain type is REAL

and where M is the following:

-- the number of digits after the decimal point if the type is real.

The format for the relation definition is the following:

```
NAME       = relation name
ATTRIBUTES = list of attributes
     NAME        = attribute name
     DOMAIN      = domain name
     CONSTRAINTS = list of constraints
     SORTED UP/DOWN/NOT SORTED
```

where CONSTRAINTS has the following format:

```
    OPERAND = L(IST OF VALUES)
              G(REATER THAN)
              S(LESS THAN)
              E(QUAL)
VALUE OR LIST OF VALUES It should be mentioned that
```

various means of entering information into the database are used. For example, some information the user is prompted for and some information, such as a query the user has to enter in a prescribed format. Although this is the case, the user should pay close attention to the information entered. Specifically, the user should avoid using the following keywords except where specified in the syntax of a query comand:

ALL	DIVIDE	JOIN	PRODUCT	UNION
AVG	FROM	MAX	PROJECT	VETO
BY	GIVING	MIN	SELECT	WHERE
COUNT	IN	ONE	SORT	$ $
DIFFERENCE	INTERSECT	OVER	SUM	

HOW TO USE THE AFIT RELATIONAL DATABASE SYSTEM * SECTION 3

3.1 Starting the System

The system is started by executing the code file DATABASE.CODE which can be located in either drive. The disk which contains this file must remain on line during the execution of the system in order to permit segment swapping.

3.2 The LOGON Procedure

Upon execution of the system, a welcome message is displayed along with the Version number of the system. Next, the user is asked to enter his identification name

by the following prompt:

Enter Identification --->

Here the user enters his assigned identification name. This name is associated with all relations created by the user and is maintained as part of the security system. The user is then asked by the following prompt whether or not his system has 1 or 2 disk drives.

DO YOU HAVE 2 OR MORE DISKS --(Y/N)-->

If 2 drives are indicated, then the domain and relation definitions are assumed to exist on the disk drive in DRIVE 1 in a file called SETUP.DATA. Otherwise, DRIVE 0 is assumed to contain the disk with SETUP.DATA. If SETUP.DATA cannot be found, the system will respond with the following prompt:

Insert proper disk in DRIVE 1 and hit return -->

The system will respond with this message until SETUP.DATA can be properly found or until the program has been halted. Once SETUP.DATA is found, the outer level prompt line is displayed as follows:

Sys Options: E(dit),R(etrieve),I(nventory),A(ttach),Q(uit)

Each option is described in detail in the following sections.

EDIT PROCEDURES * SECTION 4

To perform any edit functions, type "E" at the system level to "descend" a level into the system editor. The

following options are displayed:

```
Edit Options: I(nsert), D(elete), M(odify), C(opy),
              T(ransfer), Q(uit)
>
```

The individual Edit functions are invoked by typing the letter found to the left of the parenthesis. All Edit commands except the Copy and Transfer commands have been implemented.

4.1 I(nsert)

To insert a tuple into a relation type an "I" at the Edit level. The system immediately responds with a prompt for the relation name

ENTER RELATION NAME --->

The user should enter any combination of 1 - 15 nonblank alphanumeric characters or the following special characters: ".", "-", "_", or "/". If any other type of character is entered an appropriate error message is generated and the user is asked if he would like to correct his input. If not, the insertion process is terminated. Otherwise, the user is again prompted for the relation name. Once a relation name has been entered that is syntactically correct, it is checked as to whether or not the relation is defined. If not properly defined, the user is again asked if he would like to continue tne insertion process. If not, the process is terminated. Otherwise, the user is again prompted for a relation name.

If the relation is properly defined, the relation is

checked to see if it is attached. If not the user is given the option of attaching the relation. If the relation is attached, and the user has not logged on using the owner identification name, the following prompt is displayed:

ENTER INSERT ID --->

Here the user should enter a valid insertion identification name. This identification name was specified at the time the relation was defined. If an invalid insertion ID is entered, the user is not allowed to continue the insertion process. However, if a valid id is specified, the user is next prompted for values for each attribute in the relation. The user must enter a value for each attribute terminated by the line terminator which is a semicolon. The value entered must satisfy the domain constraints and value constraints specified for that attribute when it was defined. If not the user is given an appropriate error message and asked if he would like to correct his input. If he chooses to correct his input, the screen is cleared and the attribute name is then displayed as a prompt. Otherwise, the insertion process is terminated.

Once all the attributes are given valid values, the user is asked if he would like to insert more tuples. If more tuples are to be inserted, the user is again prompted for values for each attribute. If no more tuples are to be inserted, the entire command is displayed. The user is

then asked by the following prompt whether or not the information displayed is correct:

IS THIS THE CORRECT COMMAND --(Y/y or N/n)-->?

If the user indicates that the information displayed is indeed correct, the insertion process begins. Otherwise, it is terminated. Note, all answers to yes or no questions can be answered with either a "Y" or "y" or "N" or "n".

4.2 D(elete)

To delete a tuple from a relation type "D" at the Edit level. The system will respond with a prompt for the relation name

ENTER RELATION NAME --->

Here, the user can enter any combination of 1 - 15 nonblank alphabetic, numeric characters or the following special characters: ".", "-", "_", "/". If any other type of character is entered, an appropriate error message is generated and the user is asked if he would like to correct his input. If yes, the user is again prompted for a relation name. Otherwise, the deletion process is terminated.

Once a relation name has been entered that is syntactically correct, it is then checked as to whether or not the relation is defined. If not properly defined, an error message is printed and the user is asked if he would like to correct his input. If so, the user is again

prompted for a relation name. If the user enters a relation name that has been defined, the relation is checked to see if it has been attached. If not, a message is displayed indicating that the relation is not attached. Then the user is prompted as follows:

Attach Relation Name --(Y/y or N/n)--> ?

If the user indicates with entering a "Y" or "y" that he would like to attach the relation, the relation is then attached and the following message appears at the bottom of the screen:

ATTACHED RELATION NAME

Otherwise, if the user enters a "N" or "n", the relation is not attached and a message indicating this fact is printed at the bottom of the screen. If other than a "Y", "y", "N", or "n" is entered, the user is prompted again.

Once it has been determined that the relation is attached, the following prompt is displayed. This prompt is made only if the user has not logged on under the owner id of defined with this relation. The prompt is as follows:

ENTER DELETE ID --->

Here, the user should enter the deletion id that was specified at the time the relation was defined. If an invalid deletion id is entered, the user is not allowed to continue the deletion process. However, if a valid id is entered, the user is next prompted for values for each

attribute. This is done by displaying the attribute name and a prompt as follows:

```
ATTRIBUTE NAME
>
```

The user has the option of entering a value for an attribute. If he chooses not to enter a value for an attribute, (<cr>) should be entered. However, if a value should be entered, the following format should be followed:

Operator Value Connector

where the Operator must be one of the following:

"=" - Equal
">" - Greater than
"<" - Less than
"<>" - Not equals

and a Connector can be either "AND" or "OR". Note, a connector is optional. In addition, it should be pointed out that the blank between the operator and the value is also optional. All input may be several lines in length. Therefore, to indicate the end of input, the user must enter a semicolon. Next the input is checked to see if it is syntactically correct and if the value specified adheres to domain restrictions and constraints specified during the time the relation was defined. If there is an error in the input, an appropriate error message is displayed at the bottom of the screen. The user is then asked if he would like to correct the input. If the user chooses to correct the input the screen is cleared and the

attribute name and prompt is displayed again. Otherwise, the deletion process is aborted.

Once all input is entered the deletion command is displayed in its entirety. The user is then asked to verify the deletion operation to be performed by the following question:

IS THIS THE CORRECT COMMAND --(Y/y or N/n)-->?

Once the deletion command is verified, the operation is performed. When it is properly completed, a message is displayed indicating the fact. If the deletion command is not verified, the deletion process is terminated.

Note, all yes/no questions must be answered with either a "Y", "y", "N", or "n".

4.3 M(odify)

To modify tuples of a relation, type "M" at the edit level. The system will immediately respond with the following prompt for the name of the relation to be modified:

ENTER RELATION NAME --->

Here, the user should enter any combination of 1 - 15 nonblank alphabetic, numeric characters or the following special characters: ".", "-", "_", "/". If any other type of character is entered an error message is generated indicating this fact and the user is asked if he would like to correct his input by the following question:

WOULD YOU LIKE TO CORRECT YOUR INPUT --(Y/y or N/n)-->

If the user would like to correct the input, he should indicate this fact by entering either "Y" or "y". Otherwise, he should enter "N" or "n" and the modification process is terminated. If any other characters are entered, a message indicating that the user must enter either a "Y", "y", "N", or "n" is printed at the bottom of the screen. Also, the user is again asked if he would like to correct his input.

Once a relation name has been entered that is syntactically correct, it is checked as to whether or not the relation is defined. If not properly defined, the user is again asked if he would like to correct his input as previously described. If not, the process is terminated. Otherwise, the user is again prompted for a relation name.

If the relation is properly defined, the relation is checked to see if it is attached. If not, a message is displayed indicating that the relation is not attached. Then the user is prompted as follows:

Attach Relation Name --(Y/y or N/n)--> ?

If the user indicates by entering "Y" or "y" that he would like to attach the relation, the relation is then attached and the following message appears at the bottom of the screen:

ATTACHED RELATION NAME

Otherwise, if the user enters a "N" or "n", the

relation is not attached and a message indicating this fact is printed at the bottom of the screen. If other than a "Y/y" or "N/n" is entered, the user is prompted again.

Once it has been determined that the relation is attached, the following prompt is displayed. This prompt is made only if the user has not logged on under the owner id defined with this relation. The prompt is as follows:

ENTER MODIFY ID --->

Here the user should enter the modification id that was specified at the time the relation was defined. If an invalid modification id is entered, the user is not allowed to continue the modification process. However, if a valid id is entered, the user is next prompted for the criteria each attribute should meet to determine whether it should be modified. This done by displaying the attribute name and a prompt as follows:

ATTRIBUTE NAME
>

Here, the user has the option of entering selection criteria for an attribute. If he chooses not to enter any criteria, a (<cr>) should be entered. However, if selection criteria should be entered, the following format should be followed:

Operator Value Connector...;

where the Operator must be one of the following:

"=" - Equal
">" - Greater than
"<" - Less than
"<>" - Not equal

and a Connector can either be "AND" or "OR". Some examples of syntactically correct entries are shown as follows:

```
RANK
> =2LT OR = CAPT;
AGE
> <38;
CURRENTPCS
> = Wright Patterson OR =Eglin;
```

Note, that the blank between the operator and the value is optional. In addition, all input can be several lines in length. Therefore, to indicate the end of input, the user must enter a semicolon.

Next, the input is checked to see if it is syntactically correct and if the value specified adheres to domain restrictions and constraints specified. If there is an error in the input, an appropriate error message is displayed at the bottom of the screen. The user is then asked if he would like to correct the input. If the user chooses to correct the input, the screen is cleared and the attibute name and prompt is displayed again. Otherwise, the deletion process is terminated.

Once all selection criteria is entered, the user is prompted for the values for the attributes which are to be modified. This is done by displaying each attribute name and a prompt as follows:

ATTRIBUTE NAME=

Here, the user has the option of entering a value. If he chooses not to enter a value for the attribute, the user should enter a (<cr>). Otherwise, if a value is entered, it must adhere to the domain restrictions and constraints specifed for that attribute. If it does not meet these restrictions, an appropriate error message is displayed and the user is asked if he would like to correct his input. If not, the modification process is terminated. Otherwise, the screen is cleared and the attribute name is then displayed again.

Once, all desired values are entered, the modification command is displayed in its entirety. The user is then asked if this is the command he wishes to execute as follows:

IS THIS THE CORRECT COMMAND --(Y/y or N/n)-->?

If the user indicates that the information displayed is correct, the modification process is carried out. Otherwise, it is terminated. Note, all answers to yes/no questions must be answered with either a "Y", "y", "N", or "n".

4.4 C(opy)

This command has not yet been implemented. However, its function is as follows. The Copy command is used to copy a tuple from one relation to another. Both relations should be previously defined and attached. The Read ID

should be specified for the source relation and the Insertion ID should be specified for the destination relation.

4.5 T(ransfer)

This command has not yet been implemented. However, its suggested function is described as follows. The Transfer command is used to move a tuple from one relation, the source, to another previously defined relation, the destination. When the tuple has been moved to the destination relation, it is then removed from the source relation. To complete the Transfer process, the Deletion Id must be specified for the source relation and the Insertion id must be specified for the destination relation.

RETRIEVE PROCEDURES * SECTION 5

Type "R" at the system level to enter RETRIEVE and the following prompt line is displayed:

```
Retrieve ops: G(et),S(ave), E(dit), X(ecute),D(isplay) Q(uit)
>
```

A concept central to the operation of RETRIEVE is the command file. A command file contains one or more relational queries. The command file can be created, modified, stored on disk, retrieved from disk, and executed. The commands which can reside in a command file are described below. Several examples are provided after that.

5.1 THE QUERY COMMANDS

A. Union of two relations:

UNION relation1, relation2 GIVING relation3

where the first two relations must be union-compatible; that is, they have the same number of attributes and the ith attribute of one relation must be drawn from the same domain as the ith attribute of the other relation. Relation3 will acquire the attribute names of relation1. Relations 1 and 2 must have been attached and the READ password (if any) specified if the user does not own the relation, and relation3 must be unique.

B. Intersection of two relations:

INTERSECT relation1, relation2 GIVING relation3

where all restrictions under UNION apply.

C. Difference or relative complement of two relations:

DIFFERENCE relation1, relation2 GIVING relation3

where relation3 = relation1 - relation2. All restrictions under UNION apply.

D. Cartesian product of two relations:

PRODUCT relation1, relation2 GIVING relation3

where attribute names in relation3 will be the same as those in relation 1 and 2 except that duplicate names will be prefixed by the name of the relation it came from. Relations 1 and 2 must have been attached and the READ password (if any) specified if the user does not own

the relation, and relation3 must be unique.

E. Join of two relations:

JOIN relation1, relation2 WHERE attr1 op attr2 GIVING relation3

where attr1 is in relation1 and attr2 is in relation2, op is =, <, >. The JOIN operation is a subset of the cartesian product where the condition of membership is specified in the WHERE clause. All restrictions under PRODUCT apply.

F. Project a relation over a subset of its attributes:

PROJECT relation1 OVER attr1,attr2,...,attrN GIVING relation2

where attributes not specified in the OVER clause will be eliminated and any duplicate tuples will be eliminated. Relation1 must have been attached and the READ password (if any) specified if the user does not own the relation, and relation2 must be unique.

G. Select a subset of tuples from a relation:

SELECT ALL FROM relation1 WHERE condition GIVING relation2

where condition is a boolean predicate on the attributes of relation1 of the form a1 AND/OR a2 AND/OR a3 ... , where each aN is of the form attribute op value, where op is =, <, or >. The expression may be fully parenthesised to indicated the proper precedence of the operators, but if not then AND has precedence over OR.

One or more blanks or commas must be between each part of the command except that the left parenthesis may be flush against an item to its right, and the right parenthesis may be flush against an item to its left. Relation1 must have been attached and the READ password (if any) specified if the user does not own the relation, and relation2 must be unique.

H. Divide a binary relation by a unary relation:

DIVIDE relation1 BY relation2 OVER attr1 GIVING relation3

where relation1 is a binary relation, relation2 is a unary relation, and attr1 is an attribute of relation1 defined on the same domain as the attribute in relation2. Relation3 will be a unary relation with attribute from relation1 not attr1. Relation 1 and 2 must have been attached and the READ password (if any) specified if the user does not own the relation, and relation3 must be unique.

Examples: The following relations are used as the basis for examples:

Relation: part, Key: part no.

part	name	location	color
56	wheel	Miami	silver
78	cam	Boise	red
79	tire	Boise	black
100	seat	Dayton	green
711	fender	Miami	black
899	clock	Enon	red
1245	light	Boise	silver

Relation: shipment, Key:(part,supply

part	supply	quantity
78	4567	890
100	45	900
100	546	50
100	4567	435
899	2309	1000
899	4567	13

Relation: shippart, Key: part no.

part no.	name	location	color
78	cam	Boise	red
100	seat	Dayton	green
899	clock	Enon	red
1000	cap	Dayton	blue

Using these relations examples of the output of the above commands are shown beneath the command.

```
INTERSECT shippart, part GIVING ipart
ipart
part no.  name   location  color
78        cam     Boise     red
100       seat    Dayton    green
899       clock   Enon      red

DIFFERENCE shippart, part GIVING dpart
dpart
part no. name   location  color
1000     cap    Dayton    blue

PRODUCT shipment, dpart GIVING ppart
ppart
shipment-part  supply   quantity  dpart-part   name  location  colo
78               4567       890        1000      cap    Dayton    blu
100                45       900        1000      cap    Dayton    blu
100               546        50        1000      cap    Dayton    blu
100              4567       435        1000      cap    Dayton    blu
899              2309      1000        1000      cap    Dayton    blu
899              4567        13        1000      cap    Dayton    blu

JOIN part, shipment WHERE part no.=part no. GIVING shipment-descript
shipment-description
part-part   name  location  color  shipment-part   supply   quantity
78          cam    Boise    red          78         4567       890
100         seat   Dayton   green       100           45       900
100         seat   Dayton   green       100          546        50
100         seat   Dayton   green       100         4567       435
899         clock  Enon     red         899         2309      1000
899         clock  Enon     red         899         4567        13

PROJECT shipment-description OVER color, supply GIVING c-and-s
c-and-s
color   supply
green    45
green    546
green    4567
red      4567
red      2309
```

```
SELECT ALL FROM part WHERE location = Miami GIVING parts-in-Miami
parts-in-Miami
part no.    name   location   color
56          wheel   Miami     silver
711         fender  Miami     black


SELECT ALL FROM shipment WHERE (supply no. > 4000 AND quantity < 80
GIVING s-q
s-q
part     supply    quantity
100      4567       435
899      4567        13


PROJECT shipment OVER part, supply GIVING ps no.
ps no.
part no   supply no.
78         4567
100         45
100        546
100        4567
899        2309
899        4567


PROJECT s-q OVER supply no. GIVING s no.
s no.
supply no
4567


DIVIDE ps no. BY s no. OVER supply no. GIVING p no.

p no.
part
78
100
899
```

Each RETRIEVE command may be split between two or more lines in the command file if the split is made at the

key word. For example, some of the ways the last command in the examples above could be split is as follows:

```
DIVIDE ps no.  DIVIDE ps no. BY s no
by S no.       over supply no.
OVER supply    GIVING p no.
GIVING p no.
```

The commands may be combined in any sequence to formulate one or more queries. For example, the query "Find the colors of all parts supplied by any supplier in quantity > 500" can be expressed as:

```
SELECT ALL FROM shipment WHERE quantity > 500 GIVING T1
JOIN part, T1 WHERE part no. = part no. GIVING T2
PROJECT T2 OVER color GIVING answer
```

The query is created and executed within a command file via the commands avialable at the RETRIEVE level.

5.2 G(et)

Get a command file from disk into the workfile. A workfile is simply a command file in memory. If the current workfile is not empty then the user must decide whether or not to throwaway the current workfile before getting another.

5.3 S(ave)

Save the workfile as a command file on disk. If a previous command file was obtained using G(et) the user is asked if he wishes to save the workfile with the same name. If a file already exists on the disk with the same name, the user must decide whether or not to destroy the file on

disk before saving the workfile. If there is no room on the disk an error message is generated.

5.4 E(dit)

Typing "E" at the RETRIEVE level causes the following prompt-line to be displayed:

Edit ops: I(nsert), D(elete), B(egin), P(age), Q(uit) --->

Edit is used to create and modify command files while they are in the workfile. After execution of each Edit command except P(age) the first 20 or fewer lines of the command file is displayed on the screen, preceded by a line number.

A. I(nsert)

Insert one or more lines into the workfile. If a workfile exists then the user is prompted for a line number after which the new lines should be entered. An entry of 0 indicates before the first line and a line number greater than the last line number indicates after the last line. If a line in the file is specified, that line is displayed at the top of the screen and the user allowed to insert lines until only a return is entered. The workfile is renumbered after the insertion.

B. D(elete)

Delete a line from the workfile. The workfile is renumbered after the deletion.

C. B(egin)

Display the first 20 or fewer lines of the workfile.

D. P(age)

Display 20 or fewer lines starting at a particular line number of the workfile.

E. X(ecute)

Execute the command file in the current workfile. If syntax errors are found in the file, they are reported to the user. This causes execution to be aborted, however, the user can indicate that the syntax of the rest of the file is to be checked for syntax errors while ignoring the command in which the error occurred.

Each query of the command file is executed and the result is put into the temporary relation specified by the user in the query. If the query is determined to do nothing, such as the union of a relation with itself, then the query is not executed and this fact is reported to the user. Currently only the low-level procedure calls necessary to perform each query are output.

F. D(isplay)

Display the contents of a relation on the screen.

When quiting the RETRIEVE level, the user must decide whether or not to throwaway the current workfile, if one exists.

COMMAND SUMMARY * SECTION 6

6.1 E(dit)

"E" places the user in a level of the system called Edit. This section of the system contains commands used primarily for maintenance of relations. The options

allowed are listed as follows:

1. Insert -- allows the user to add tuples to a relation

2. Delete -- allows the user to remove tuples from a relation

3. Modify -- allows the user to make changes to tuples of a relation

4. Copy -- allows the user to insert tuples from relation into another

5. Transfer -- allows the user to move tuples from relation into another relation

See section 4 for more detail.

6.2 R(etrieve)

This command allows the user to formulate and execute relational queries on the database relations, and display the results. The user is allowed to formulate queries that perform the select, project, or join operation. In addition, the user can perform the union, intersect, divide, and difference operations. A workfile is used to hold queries, and commands are included perform various functions on this workfile.

6.3 I(nventory)

Typing "I" at the system level will cause a list of the relations which have been defined and attached to be displayed on the CRT. Note, all information but the security identification names are listed.

6.4 A(ttach)

This command allows the user to have access to a relation. Currently, only an attach flag is set in the

relation definition. However, the attach command should request that some sort of security id be specified. This security id could be the read id.

6.5 Q(uit)

This command allows the user to exit from the database system. During this process, all relation and domain definitions are written out to disk, and a message is displayed when this process has completed.

CHANGING THE SYSTEM * SECTION 7

Please refer to sections 3.3.1 and 3.3.2 of the UCSD Pascal Version II.0 reference manual and current program listings before trying to change the system.

The program is currently divided into a main body segment, four segment procedures and a unit. Each segment is contained in a dummy program in order to permit seperate compilation. The unit contains all types, variables, and procedures which are global to more than one segment. Thus by including the unit in each dummy program, access is allowed to those elements. The format for each segment procedure is:

```
PROGRAM dummy-name;
USES COMMON;  (*the unit is named COMMON*)
SEGMENT PROCEDURE name(parameter-list);
  Local types, variables, and procedures;
BEGIN
  body of name;
END; (*name*)
BEGIN
END.  (*dummy name*)
```

Since the segments are seperately compiled, the parameter list of the segment procedure, must contain all

global variables accessed or modified by the procedure. The program or segment procedure whcih calls each segment procedure must have a dummy segment procedure with the same name, so that it may compile properly. The format for the main body segment is:

```
PROGRAM main;
USES COMMON;
  local labels,types,constants, and variables;
SEGMENT PROCEDURE name1( --- );
BEGIN
END; (*name1*)
SEGMENT PROCEDURE name2( --- );
BEGIN
END; (*name2*)
.
.
.
other local procedures;
BEGIN
  body of main;
END. (*main*)
```

The format for segment procedures which use other segments is the same as the previous format for a segment procedure except dummy segment procedures ae included as local procedures.

Each segment procedure has a particular segment number from 11 to 15 associated with it. The main body segment has number 1 and the unit has number 10. Other numbers are for Pascal use only. The way numbers are assigned to the segment procedures is in first compiled, first numbered order. Thus, in the above format for the main body segment name1 would be assigned 11, name2 assigned 12, etc. Therefore, in each dummy program used to define a segment procedure an appropriate number of dummy

segments must exist before the defined segment to ensure that the segment count is the same. Thus, for example, the format for segment name2 would be:

```
PROGRAM dummy name;
USES COMMON;
SEGMENT PROCEDURE dummy name1;
BEGIN
END; (*dummy name1*)
SEGMENT PROCEDURE name2( --- );
  local labels, types, etc.
BEGIN
  body of name2
END; (*name2*)
BEGIN
END.  (*dummy name*)
```

The unit -- COMMON -- is compiled and placed in the system library using the librarian program, LIBRARY.CODE. (See section 4.2 of the UCSD Pascal manual.) When each program is compiled, the unit is retrieved from the system library and used in the program, however, each program must still be linked with the system library in order to bind the external variable and procedure references into the unit. After each program is compiled and linked, then the librarian may be used to put all the segments together into one code file. Each code file containing the segment is retrieved and linked into the proper space using its assigned segment number into the overall file.

If a particular segment, including the main body segment, is to be changed then the steps to be followed are:

1) Change the source code for the segment.
2) Compile the program containing the segment.
3) Link the code file to the system library.
4) Using the librarian create a new overall

file passing all unchanged segments to the new file and linking in the changed segment.

If the unit has to be changed then after compiling it and placing it into the system library, each program must be recompiled, linked to the system library, and then put together with the librarian.

AFIT/GCS/EE/82D

APPENDIX C

THE CONTINUED DESIGN AND IMPLEMENTATION OF A RELATIONAL DATABASE SYSTEM

by

Linda M. Rodgers
2Lt USAF

Graduate Computer Systems

December 1982

APPENDIX C

THE CONTINUED DESIGN AND IMPLEMENTATION OF A RELATIONAL DATABASE SYSTEM

ABSTRACT

The AFIT Relational Database System is a relational database management system currently being implemented on the LSI-11 using the UCSD Pascal Operating System. The system presently supports a high level relational algebraic query language which will provide a means for users to manipulate and query the database system. Presently, the system allows the user to define domain and relation definitions and build and optimize queries.

The goal of this thesis effort is to continue the development of this training tool for teaching relational database concepts and for general research purposes.

INTRODUCTION

Database technology has become one of the most rapidly growing fields in computer science as a result of the increase of information available and the increasing demand for efficient methods of maintaining and accessing this information. As a consequence of such an increasing demand for database technology, an increasing need for those who are skilled in this field exists. However, new teaching aids are needed in order to train the people who are to meet this demand for experienced individuals in the

database field. Specifically, these tools should provide an opportunity for the student to apply concepts learned from the literature in a laboratory environment. Thus, at the Air Force Institute of Technology, such a tool was proposed to be implemented. Particularly, Dr. Thomas Hartrum on the faculty of the AFIT/EN Electrical Engineering Department proposed that such a tool be developed as a master's degree thesis.

BACKGROUND

In the design of the database management system, two views of the database exists. One view is the logical view of the database. The second view is the actual physical representation of the database system. The logical view of the database system is basically a view of the data that the user sees. However, this view is eventually mapped into the physical storage device. Generally, the approach to the development of this relational database system is partitioned into two phases. The first phase includes the design and implementation of the logical database system, that is, how the user defines the data as relations and manipulates the data. The second phase is concerned with the mapping from the physical storage device to a form which can be manipulated in main memory.

INITIAL DESIGN CONSIDERATIONS

The first phase of development was initiated by 2nd Lt. Mark A. Roth. During this phase several decisions were made concerning the system and language to be used for

development. Specifically, it was decided that the hardware used should be readily available for AFIT students and that the system should provide some means of interactive capability. At the time development began, these requirements were fulfilled by designing the system so that it could be used on the INTEL 8080 microcomputer. The language to be used was UCSD Pascal. The reasons were it was still a widely used implementation among microcomputers which would allow it to be brought up on another system if necessary. In addition UCSD Pascal was also chosen because it allows program segmentation with the result that it can handle large programs conveniently by providing separate compilation and segment swapping.

ORIGINAL DESIGN OF THE DDL AND THE DML

The general approach to the design of this relational database system was to design at a high level, the entire data base system and implement those modules which dealt with how the user would define his view of the database and query the database.

THE DATA DEFINITION FACILTIY

The function of the data definition facility is to allow the user to define domain and relation definitions through the centralized control of the database administrator. To define a domain definition the following information has to be entered:

```
NAME = domain name
TYPE = CHAR/TEXT/INTEGER/REAL
```

N,M = maximum number of characters in a value of a particular domain

The define a relation definition the following information has to be entered:

NAME = relation name
ATTRIBUTES = list of attributes
  NAME = attribute name
  DOMAIN = domain name
CONSTRAINTS = list of constraints
SORTED UP/DOWN/NOT SORTED

THE DATA MANIPULATION FACILITY

The function of the data manipulation facility is to provide the user with a means of updating and querying the database. The approach to the design of the data manipulation language was to consider it a language that would allow the user to perform edit functions and a language that would allow the user to perform retrieval functions. Consequently, the data manipulation language consists of those commands which provide for the insertion, deletion, and modification of relations and those which provide the user with the capability to query the database. The commands which provide for the update of relations are given as follows:

INSERT
MODIFY
DELETE
COPY
TRANSFER

The retrieval commands are based on the relational algebraic language developed by Codd. They are given as follows:

INTERSECT
DIFFERENCE

UNION
PRODUCT
JOIN
PROJECT
SELECT
DIVIDE

An interesting feature included in the design and implementation of the DML facility is the concept of query optimization. Specifically, Mark A. Roth provides the user with the capability to formulate query command files which consists of a number of queries which are executed in "batch" mode. This allows the query commands to undergo an optimization process which was formulated by Miles Smith and Philip Chang (Ref 3).

As stated, the design and implementation of this system is approached in two phases. The first phase consists of the design and implementation of the DDL and the DML. At the end of Mark Roth's thesis effort, this was completed. However, the means of accessing the data from the physical storage unit still needed to be implemented.

THE DESIGN OF THE ACCESS METHOD

To provide a means for designing and implementing the access method, Mark Roth suggested an access method formulated by Theo Haerder called the generalized access path method. Specifically, this access method consists of a tree structure which serves as an index to a set of terminal nodes (leaf nodes) which contain tuple identifiers (TID's). These are the addresses of tuples which contain an attribute with a certain value. Generally, a tree structure exists for each domain defined in the database

system. Consequently, a means exists for accessing a tuple based on any attribute value in the tuple (Ref 1).

IMPLEMENTATION OF THE ACCESS METHOD

The design and partial implementation of this access structure was accomplished. During the design phase several decisions concerning the B-tree structure had to be made. These are given as follows:

1. the file format for the relation file
2. the file format for the B-tree structure
3. the number of physical blocks which would make up a B-tree node and leaf node
4. the fanout ratio for a B-tree node

The decisions concerning file formats and blocking factors were based on tradeoffs between memory space, ease of programming and processing time considerations. Specifically, it was decided that files would consist of a varying number of blocks where each block may consist of a number of fixed length records (Ref 2). The decisions concerning the design of the tree structure were based on tradeoffs between ease of programming and memory space considerations. Specifically, each node was considered to be a block which consists of 512 bytes on the LSI-11. To make it easier to implement, each record in a block was a fixed length so that the beginning and end of each record was easy to identify. To potentially save memory space and processing time, the fanout ratio, which is the maximum number of records allowable in a node could vary depending

on the maximum length of a domain value. This was in lieu of all the B-trees having the same fanout ratio. This has the potential to waste space because more records may actually be able to fit in a node than the fanout ratio specified. If the fanout ratio is fixed, then a new node must be used in order to insert a record that can potentially fit in an existing node, thus increasing the level of a B-tree. This may in turn increase processing time, because more disk accesses may have to be done in order to traverse the tree to get to the leaf nodes.

SUMMARY

As can be seen, the design and implementation of this database system addresses several key aspects of a relational database system. Specifically, it provides a means for specifying domain and relation definitions and a means for specifying and optimizing query commands. There are still several aspects of the development of the AFIT Relational Database System to be considered. Specifically, the low level access mechanism needs to be completed.

References

1. Haerder, Theo. "Implementing a Generalized Access Path Structure for a Relational Database System," ACM Transaction on Database Systems, 3: 285-298 (1978).

2. Rodgers, Linda M. "The Continued Design and Implementation of a Relational Database System," (Thesis) School of Engineering, Air Force Institute of Techology, Wright Patterson AFB, Ohio 1982.

3. Smith, Miles and Philip Yen-Tang Chang. "Optimizing the Performance of a Relational Algebra Database Interface." Communications of the ACM, 18: 568-579 (1975).

VITA

Linda M. Rodgers was born May 21, 1959 in Bitburg, Germany. She graduated with a B.S. in Computer Science in 1981 from the Georgia Institute of Technology. On December 17, 1982, she graduated with an M.S. in Computer Systems from the Air Force Institute of Technology.

Permanent Address: 101 Seminole Circle
Niceville, Florida 32542

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFIT/GCS/EE/82D-29	2. GOVT ACCESSION NO. AD-A124927	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) THE CONTINUED DESIGN AND IMPLEMENTATION OF A RELATIONAL DATABASE SYSTEM		5. TYPE OF REPORT & PERIOD COVERED 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Linda M. Rodgers 2Lt. USAF		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Air Force Institute of Technology (AFIT/EN) Wright-Patterson AFB, Ohio 45433		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS		12. REPORT DATE 17 December 1982 13. NUMBER OF PAGES 178
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

Approved for public release: IAW AFR 190-17.

LYNN E. WOLAVER
Dean for Research and Professional Development
Air Force Institute of Technology (AU)
Wright-Patterson AFB OH 45433

4 JAN 1983

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Relational Databases
Computer Science
Access Structure Design

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The Roth Relational Database System is a database system designed and implemented for teaching and research purposes at the Air Force Institute of Technology. The system was originally designed and partially implemented by 2Lt Mark A. Roth in 1979 and continued design and implementation has been accomplished by James Mau. During this thesis effort, an investigation into the design and implementation of the Roth Relational Database System is made in order to continue the design and

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

implementation of the AFIT Relational Database System. Additional research was done to assess the feasibility of suggestions made by Roth concerning the continued development of the system. Specifically, an investigation was made concerning Theo Haerder's access structure and an investigation was made concerning the interface between the editor and the low level access mechanism.

With this research accomplished, a top down structured design was completed for the edit functions and the low level access structure. Once this was accomplished an algorithm was developed, implemented and tested for the insertion and deletion of values into a B-tree.

2) Compile the program containing the segment.
3) Link the code file to the cyrter library.
4) Using the librarian create a new overall